SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2017, Vol. 60 092113:1–092113:16
doi: 10.1007/s11432-017-9112-2

Lightweight fault localization combined with fault context to improve fault absolute rank

Yong WANG[1,2], Zhiqiu HUANG[1*], Yong LI[1] & Bingwu FANG[1]

[1]*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing* 210016*, China;*
[2]*College of Computer and Information, Anhui Polytechnic University, Wuhu* 241000*, China*

Received January 3, 2017; accepted May 31, 2017; published online July 27, 2017

Abstract Lightweight fault localization (LFL), which outputs a list of suspicious program entities in descending order based on their likelihood to be a root fault, is a popular method used by programmers to assist them in debugging. However, owing to the nature of program structures, it is impossible for LFL to always rank the root faulty program entity at the top of a ranking list. Recently, Xia et al. noted in their study that programmers inspect the first top-K-ranked program entities outputted by an LFL tool in sequence. Therefore, it is of practical significance to further improve the absolute rank of those buggy programs by using LFL if the root fault is ranked higher. To solve this issue, we propose a new LFL combined with a fault context to improve the fault absolute rank. We conduct experiments in which we apply our proposed approach to seven Siemens benchmark programs. The results show that by combining the suspiciousness scores of program entities with their fault-context suspiciousness scores that are based on an LFL called Dstar, our approach can improve the fault absolute rank with an effectiveness rate of 35.7% for 129 faulty versions from the seven benchmark programs. It should be noted that our approach can obtain an average improvement of 65.18% for those improved programs to which LFL can be effectively applied, and that there were improvements to seven top-ranked root faults of buggy programs.

Keywords fault context, fault localization, program spectrum, absolute rank, debugging

Citation Wang Y, Huang Z Q, Li Y, et al. Lightweight fault localization combined with fault context to improve fault absolute rank. Sci China Inf Sci, 2017, 60(9): 092113, doi: 10.1007/s11432-017-9112-2

1 Introduction

The use of lightweight fault localization (LFL) techniques, which use instrumentation devices that have a low computational overhead and good scalability (as they can still produce good results in large code bases [1]), has been shown to be a very practical and efficient fault localization strategy. The main idea behind LFL is as follows: program entities (such as methods, statement blocks, statements, or predicates) that are primarily executed by failed test cases are more likely to be root faults than program entities that are primarily executed by successful test cases. Therefore, these techniques output a suspicious program entity list, in which program entities are ranked based on their suspiciousness scores, and the list is given to programmers to aid them in identifying the root fault of program failures.

* Corresponding author (email: zqhuang@nuaa.edu.cn)

© Science China Press and Springer-Verlag Berlin Heidelberg 2017

Empirical studies (e.g., [2–5]) show that LFL approaches can be effective in helping programmers locate bugs. However, to determine whether those LFL techniques actually help programmers, Parnin and Orso [6] performed a pioneer user study and noted that many developers do not consider LFL tools to be useful if the root cause is not listed early in the suspicious program entities rank list. Recently, some studies revisited the usefulness of LFL by conducting empirical or survey research [7–10]. Whether they regarded LFL with optimism or pessimism, they all highlighted the fact that researchers should continue to innovate and design more accurate fault-localization tools to ensure that root faults appear in higher positions in rank lists. Therefore, in order to apply LFL in practice, it is important to know how to further improve a root fault's absolute rank.

Although past studies have shown that LFLs are sufficiently effective for a large number of cases, LFL techniques are not sufficiently effective for many other cases. In such cases, the root fault is often ranked low in the fault rank list [11]. It is generally known that LFL techniques compute the statistical difference between failed and passed spectra to generate a program entity's suspiciousness score. Intuitively, those cases in which the root fault is ranked low in the rank list are not suitable for LFL; conversely, we could further improve the absolute rank if a root fault is ranked higher in the rank list. Therefore, we mainly focus on further improving the absolute rank for those buggy programs if the application of an LFL technique to the buggy programs is effective.

Therefore, the key research question of this paper is as follows: How can we further improve the absolute rank using LFL techniques for buggy programs if the application of an LFL technique to the buggy programs is effective?

As is well known, LFL techniques focus on computing the suspiciousness score for individual program entities, but they ignore the fault context of those program entities. We model the fault context's suspiciousness using a lightweight statistical technique, and combine the program entity's suspiciousness with its fault-context suspiciousness to balance the suspiciousness score among program entities, which can further improve the absolute rank. In this paper, we propose a novel LFL approach based on two suspiciousness ranks: the suspiciousness of a program entity and the suspiciousness of a program entity's fault context. Informally, the fault context of a program entity consists of other suspicious program entities that were covered in the same failed execution except the program entity. Intuitively, the higher the suspiciousness of a program entity, the lower the suspiciousness of its fault context and the higher will be the ranking of this program entity.

The main contribution of our research is twofold: (i) to the best of our knowledge, the work is the first to further improve the fault absolute rank based on LFL techniques; (ii) we verified our approach by experimenting on seven Siemens programs from the Software-artifact Infrastructure Repository (SIR) [12], and we compared the results with those obtained for four other LFL techniques, namely, Tarantula [5], Jaccard [4], Ochiai [4], and Dstar [13]. The empirical results show the potential of our proposed approach.

The rest of this paper is organized as follows. In Section 2, we describe the LFL technique and our motivation. In Section 3, we present our technique, while in Section 4, we present our empirical research. We discuss the results in Section 5, and in Section 6, we conclude the paper and present future work.

2 Background and motivation

In this section, we summarize the relevant background regarding LFL techniques, and discuss our motivation for combining the suspiciousness ranks of program entities with their fault-context suspiciousness rankings to further improve the absolute rank.

2.1 Lightweight fault localization

LFL techniques are statistical fault-localization techniques that calculate the likelihood of a program entity being the root cause of program failure for each program entity in buggy programs [2]. They exploit the program spectrum, which includes coverage information regarding which component is covered in each execution, from failed and successful software executions. Collecting program spectra using an

Table 1 Lightweight fault localization

Name	Metric
Jaccard	$\frac{n_{11}(s)}{n_{01}(s)+n_{10}(s)+n_{11}(s)}$
Tarantula	$\frac{(n_{10}(s)+n_{00}(s))\times n_{11}(s)}{(n_{01}(s)+n_{11}(s))\times n_{10}(s)+(n_{10}(s)+n_{00}(s))\times n_{11}(s)}$
Ochiai	$\frac{n_{11}(s)}{\sqrt{(n_{01}(s)+n_{11}(s))\times(n_{11}(s)+n_{10}(s))}}$
Dstar	$\frac{(n_{11}(s))^*}{n_{01}(s)+n_{10}(s)}$

		Coverage (0=executed,1=unexecuted)												Normalized
No.BB	Prog1(a,b)	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	Dstar scores
1	if(a<10&&b<10)	1	1	1	1	1	1	1	1	1	1	1	1	0.22
2	result=$a-b$;//bug if(result>0)	1	1	1	1	0	1	1	1	1	1	1	1	0.25
3	print("positive");	1	0	0	0	0	1	1	0	0	1	1	1	0.01
4	else if(result==0)	0	1	1	1	0	0	0	1	1	0	0	0	0.33
5	print("zero");	0	1	0	0	0	0	0	0	1	0	0	0	0.03
6	else print("negative")	0	0	1	1	0	0	0	1	0	0	0	0	0.15
7	else print("invalid input");	0	0	0	0	1	0	0	0	0	0	0	0	0.00
Program results (0=successful,1=failed)		1	1	1	1	0	0	0	0	0	0	0	0	

Figure 1 (Color online) A buggy program and its program spectrum.

instrumented program version is a lightweight analysis method, and a program spectrum provides a dynamic view of the behavior of the system under test [14]. Because LFL techniques only register whether or not a program entity is covered during a certain run, binary flags ("0"/"1") can be used for each program entity, the program spectra of which are also called hit spectra [14].

The program spectra between successful and failed runs are given to calculate the suspiciousness score for each entity. All program entities are then ranked in descending order based on their suspiciousness score to aid programmers in performing a manual check.

The key for LFL is the similarity metric used to compute the suspiciousness. There are several similarity metrics [15]. Table 1 shows the similarity metric of four well-known LFL approaches: Jaccard, Tarantula, Ochiai, and Dstar. Given the program spectrum and the result vector, which represent the result of each execution, we calculate n_{00}, n_{01}, n_{10}, and n_{11}. For each program entity s, where $n_{00}(s)$ is the number of successful executions in which program entity s is not covered, $n_{01}(s)$ is the number of failed program executions in which entity s is not covered, $n_{10}(s)$ is the number of successful executions in which program entity s is covered, and $n_{11}(s)$ is the number of failed executions in which program entity s is covered. The suspiciousness metrics of the above mentioned LFL techniques are defined in Table 1.

2.2 Motivation example

In this section, we use the example shown in Figure 1 to illustrate our ideas. The example program contains a bug in block 2, which calculates the sum of two integers. The correct statement should be result= $a+b$. However, we accidentally write the correct statement as result= $a-b$. There exists a set of 12 test cases, of which 4 test cases (t_1, t_2, t_3, t_4) fail and 8 test cases (t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}) are executed successfully. Block-hit spectra are shown in the columns, where a "1" indicates that the basic block is covered at least once and a "0" indicates that the basic block is not covered by the execution; the success/fail status is shown in the bottom row. Those columns representing the failed test cases are highlighted. The Dstar LFL tool is used to compute the suspiciousness score of a basic block b. Each basic block b is assigned a normalized suspiciousness score between 0 and 1 to represent the likelihood

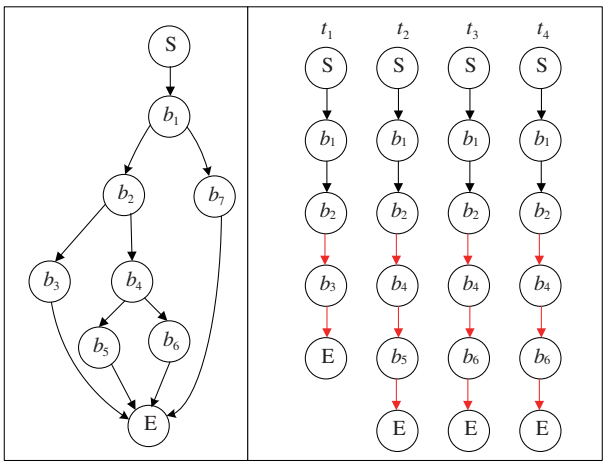

Figure 2 (Color online) Control flow graph and failed execution traces.

that b is the root fault. For instance, in the example shown in Figure 1, b_4 has the highest score (0.33) among the seven basic blocks, which is higher than that of the root fault b_2.

As mentioned in the above section, an LFL technique such as Dstar only focuses on computing the suspiciousness for each program entity independently according to some criteria that measure the correlation, which can be viewed as the suspiciousness score, between a program entity and failures, after which it then ranks them. Based on the fault/failure model, faulty program entities are triggered in an execution, and the infected states are propagated, causing failure. Therefore, whether or not a failure is observed is dependent on the fault that is triggered and its context. For example, b_2 was triggered in all test cases except t_5; however, we only observe failure in t_1, t_2, t_3, and t_4 because of its different contexts in different executions. Hence, to further improve the absolute rank, we should model the suspiciousness of both the program entity and its fault context.

Generally, even if we know the fault location in a program, it is extremely difficult to determine the corresponding fault context in which a failure(s) could be triggered. As a matter of fact, we cannot know the location of a fault in practical complex programs. Considering that program entities interact with each other through complex control and data dependency, extracting a basic block's fault context should involve the use of heavyweight fault-localization techniques. Considering the cost of heavyweight fault localization, we only use hit spectra to model the suspiciousness of the fault context in our LFL technique. To simplify our LFL, we informally define a program entity's fault context as a set of entities consisting of other suspicious program entities that were covered in the same failed execution. In this example, we can ignore basic block b_7 to compute all fault contexts because its suspiciousness score is zero.

The control flow graph and failed execution traces for the buggy program are shown in Figure 2, where b_2 is in the failed execution trace executed by the failed test case t_1, its fault context in this execution is $\{b_1, b_3\}$, and the suspiciousness score of its fault context is defined as the sum of the suspiciousness scores of b_1 and b_3. Similarly, for the program executed in t_2, t_3, and t_4, its fault context is $\{b_1, b_4, b_5\}$, $\{b_1, b_4, b_6\}$, and $\{b_1, b_4, b_6\}$, respectively, and the suspiciousness score of its fault context can be defined as the sum of the suspiciousness scores of all basic blocks in the program entity's fault context. For basic block b_2, there exist four suspiciousness scores of its fault context. Intuitively, because a program entity ranking increases as the suspiciousness score of its fault context increases, we should choose the minimum suspiciousness of the four suspiciousness scores as the fault-context suspiciousness score of b_2.

In the example, b_4 is ranked higher than root fault b_2 based on the suspiciousness score of the program entity. From the execution traces shown in Figure 2, we find that b_4 was influenced by b_2 and that its fault context includes b_2. Conversely, b_2 can trigger a failure based on a fault context that does not contain b_4. Thus, we can use the fault-context suspiciousness rank list to balance the suspiciousness scores between b_2 and b_4. The suspiciousness score of the fault context of b_2 can be calculated as follows: $\rho_c(b_2) = \min(\rho_b(b_1) + \rho_b(b_4) + \rho_b(b_5),\ \rho_b(b_1) + \rho_b(b_3),\ \rho_b(b_1) + \rho_b(b_4) + \rho_b(b_6)) = \min(0.58, 0.23, 0.70) =$

Table 2 Suspiciousness of BBs and their contexts

Basic block No.	Dstar		Fault context		FCLFL-Dstar	
	ρ_b	r_b	ρ_c	r_c	$r_b + r_c$	Rank
b_1	0.22	3	0.26	2	5	2
b_2	0.25	2	0.23	1	3	1
b_3	0.01	6	0.47	3	9	4
b_4	0.33	1	0.50	4	5	2
b_5	0.03	5	0.80	5	10	6
b_6	0.15	4	0.80	5	9	4
b_7	0.00	–	–	–	–	–

0.23. In this formula, $\rho_c(b_2)$ is the fault-context suspiciousness score of b_2, and $\rho_b(b_4)$ is the suspiciousness score of b_4. Similarly, the suspiciousness score of the fault context of b_4 can be calculated as $\rho_c(b_4) = \min(\rho_b(b_1) + \rho_b(b_2) + \rho_b(b_5),\ \rho_b(b_1) + \rho_b(b_2) + \rho_b(b_6)) = \min(0.50, 0.62) = 0.50$.

The suspiciousness score of the fault context of b_2 is clearly lower than that of b_4. In other words, according to the suspiciousness score of its fault context, b_4 may be influenced by others, thus obtaining a higher program-entity suspiciousness score. For the example, the information related to its fault localization is listed in Table 2.

3 Our approach

3.1 Problem settings

Let $P = \{c_1, c_2, \ldots, c_m\}$ be a buggy program, where c_i is a program component contained by P. Let $\Gamma = \{t_1, t_2, \ldots, t_n\}$ be a test suite, Γ be divided into two parts Γ_p and Γ_f, Γ_p be a test suite associated with successful executions, and Γ_f be a test suite that can trigger program failures. In our motivation example, for instance, $\Gamma_f = \{t_1, t_2, t_3, t_4\}$ and $\Gamma_p = \{t_5, t_6, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}\}$. Similar to existing LFL works, our approach first collects the program spectra, after which it then assesses the suspiciousness scores of program entities and their fault contexts. Finally, it generates a ranked program component list for P in descending order based on the two suspiciousness ranks.

3.2 Framework

Our LFL algorithm combined with Fault Context, called FCLFL, is illustrated in Algorithm 1. The approach has three major steps. (1) Suspiciousness measurement for program entities. In this step, we first generate coverage matrix A, using execution traces collected by the executing test suite, and result vector e, which represents the information regarding successful or failed executions. We then use the suspiciousness metric to calculate the suspiciousness score for every program entity (lines 1–14). (2) Suspiciousness measurement for fault contexts. In this step, we first normalize the suspiciousness scores of program entities. Then, we obtain the fault context for each program entity and calculate its suspiciousness (lines 15–23). (3) Fault rank list generation. In this step, we combine the two above-mentioned suspiciousness scores to generate a new fault rank list (lines 24–26). Next, we discuss the three main steps.

3.3 Suspiciousness measurement for program entities

In this step, we first run an instrumented version of program P with a test suite Γ, and gather a collection of the execution traces. Then, these execution traces are converted to form a program hit-spectrum, which is represented as coverage matrix A at the block level. Result vector e is then collected using the test oracle.

We calculate n_{00}, n_{01}, n_{10}, and n_{11} for each program entity j based on A and e, where $n_{00}(j)$, $n_{01}(j)$, $n_{10}(j)$, and $n_{11}(j)$ are defined as in the above section. Based on those values, the suspiciousness score

Algorithm 1 FCLFL algorithm

Input: Program P, test suite Γ, and suspiciousness metric ρ_b;
Output: fault rank list Ω;

```
//step 1: suspiciousness measurement for program entities;
(A, e) ← Run_program (P, Γ);
N ← |Γ|;
M ← getNumberofComponents(P);
Ω ← ∅;
for i = 0 to M do
    {n00 (j) , n01 (j) , n10 (j) , n11 (j)} ← 0;
end for
for i = 0 to M do
    set values (n00 (j) , n01 (j) , n10 (j) , n11 (j));
end for
for i = 0 to M do
    S[i] ← ρb (n00 (j) , n01 (j) , n10 (j) , n11 (j));
end for
//step 2: suspiciousness measurement for fault contexts;
// normalize S to make ∑ si = 1;
normalize(S);
for j = 0 to M do
    // get fault context of program entities j;
    fc_j = get_fault context (A, e, j);
    //calculate suspiciousness score for each program entity's fault context;
    S_fc [j] = ρc (fc_j, S);
end for
//step 3: fault rank list generation;
Ω ← sort (S, S_fc);
return Ω.
```

($\rho_b(j)$) for program entity j is calculated given a specific metric, and ($\rho_b(j)$) can be viewed as $p(j$ is a fault) if we normalize those suspiciousness scores to make $\sum \rho_b(j) = 1$.

Let ρ_b be defined such that the higher the ρ_b that a program entity j has, the more likely will j trigger failures. In the field of fault localization, there are several suspiciousness metrics that could be used to compute ρ_b. We choose Dstar as our benchmark to calculate the suspiciousness of program entities. Dstar was proposed by Wong et al., where the possibility of a program entity being at fault is (i) proportional to the number of failed tests that executed it, (ii) inversely proportional to the number of successful tests that executed it, and (iii) inversely proportional to the number of failed tests that did not execute it. More importantly, (i) is the soundest intuition and should have a higher weight than that of (ii) and (iii). Dstar is defined as shown in (1):

$$\rho_b^{\text{Dstar}}(j) = \frac{(n_{11}(j))^*}{n_{01}(j) + n_{10}(j)}. \tag{1}$$

In (1), $*$ is a power of $n_{11}(j)$ whose value is equal to or greater than 1. In [13], the experimental results showed that Dstar outperformed all of the other LFL techniques with which it was compared. They examined the relationship between the effectiveness of Dstar and the value of $*$, and they found that the effectiveness increases as the value of $*$ increases, before leveling off after $*$ reaches a critical point. We set $*$ equal to 2 in our experimental study.

3.4 Suspiciousness measurement for fault contexts

We now give a formal definition of a program entity's fault context. Let $\text{ec}_i = \{e_1, \ldots, e_j, \ldots, e_k\}$ be the covered set of program entities for a failed execution i. According to the definition of a program spectrum, program entities $e_1, \ldots, e_j, \ldots, e_k$ in the failed execution t_i are executed more than once. The fault context of e_j is a set of all program suspicious entities, except for e_j, that appear in the failed execution. For a failed execution, the fault context of e_j and its suspiciousness score are as shown in (2) and (3), respectively:

$$\text{fcontext}_c(e_j, t_i) = \text{ec}_i / e_j, \tag{2}$$

$$\rho_c(e_j, t_i) = \Sigma\rho_b(\mathrm{ec}_i(k)). \tag{3}$$

In particular, the suspicious entities of a program can be specified by programmers. Let us suppose that a program entity is a suspicious entity if the program entity is ranked in the top three based on the Dstar tool. For our motivation example, b_4, b_2, b_1 are regarded as suspicious program entities; thus, we ignore b_3, b_5, b_6, b_7 when computing the fault context. In this case, $\mathrm{fcontext}_c(b_2, t_2) = \{b_1, b_2, b_4\}/b_2 = \{b_1, b_4\}$.

If there is more than one fault context for e_j in Γ_f, we are interested in the minimum value among fault contexts for e_j. Hence, the fault context of e_j is defined as shown in (4), and the suspiciousness of the fault context of e_j in those failed runs can be calculated, as shown in (5):

$$\mathrm{fcontext}_c(e_j) = \{\mathrm{ec}_i/e | i \in \text{set of failed executions}\}, \tag{4}$$

$$\rho_c(e_j) = \min(\rho_b(e_j, t_i) | i \in \text{set of failed executions}). \tag{5}$$

Taking the motivation example as an illustration, suppose that the top-all program entities in the rank are suspicious based on the Dstar tool; there are three different fault contexts for b_2 (because there exist two identical fault contexts in running t_3 and t_4). Then, considering b_2 as the root cause of failures, different suspiciousness values for b_2 are probably influenced by b_2. Therefore, we choose the minimum suspiciousness value as the suspiciousness of the fault context for b_2, which could be used for comparison with the suspiciousness of the fault context of other program entities because the fault context is minimally impacted by b_2. Conversely, considering that b_4 is not the root cause of failures, there exist two different fault contexts for b_4 (similarly, we retain one of the two identical fault contexts to run t_3 and t_4). We choose the minimum value among the two suspiciousness scores of the fault context for b_4. Because of the effect of the root cause of failures, the fault contexts for b_4 all contain the root cause b_2. Note that our goal is to improve the absolute rank for those buggy programs for which the use of LFL could be effective. Therefore, the root cause of failures (b_2) should have a higher rank. Thus, the suspiciousness value of the fault context of b_4 should have a larger rank of suspiciousness scores of the fault context.

To measure the suspiciousness of a fault context, we first normalize the suspiciousness score for each suspicious program entity, which requires that we make $\sum \rho_b(j) = 1$. We then generate fault contexts for each program entity in each failed execution. Finally, according to the definition of suspiciousness of a fault context, the suspiciousness score of the fault context for each program entity is computed.

3.5 Fault rank list generation

After steps 1 and 2 in running FCLFL algorithm, we get the suspiciousness scores for program entities and their fault contexts. Two fault rank lists can be generated according to the two suspiciousness scores. In this step, we further synthesize a new fault rank list. Intuitively, if a program entity s is the root fault, then the suspiciousness score of its fault context can potentially be lower. On the other hand, if the suspiciousness score of the fault context of program entity s is higher, the suspiciousness score of s will be lower. In other words, it is impossible for s to be the root fault. Therefore, we can balance the suspiciousness between s and its fault context. The higher the value of $\rho_b(s)$, the lower the value of $\rho_c(s)$ and the higher will be the program entity rank in the new fault rank list.

To generate a new fault rank list, we first generate the two fault rank lists for a buggy program, r_b and r_c, where r_b is in non-ascending order of ρ_b and r_c is in non-descending order of ρ_c. The new fault rank list r is generated by combining the two rank lists as follows. Let b_i and b_j be two suspicious program entities. Assuming that b_i has ranks r_b^i and r_c^i, and b_j has ranks r_b^j and r_c^j, in the new fault rank list r, b_i is ranked higher than b_j if and only if $r_b^i + r_c^i \leqslant r_b^j + r_c^j$.

3.6 Space and time complexity

Our FCLFL approach uses only the program spectra and result error vector to calculate a program entity's suspiciousness and a fault context's suspiciousness. The time and space complexity of our approach is

Table 3 Subjects used for empirical studies

Program	LOC	#Blocks	#Fault version	#Test cases	K	Description
print_tokens	478	122	5	4130	10	Lexical analyzer
print_tokens2	399	135	10	4115	10	Lexical analyzer
replace	512	153	32	5542	10	Pattern matcher
schedule	292	73	9	2650	8	Priority scheduler
schedule2	301	77	9	2710	8	Priority scheduler
tcas	141	23	41	1608	3	Aircraft control
tot_inf	440	74	23	1052	8	Info measure

similarly analyzed, as for other LFL techniques: With the same problem settings ($|\Gamma_f|$ is the number of failed runs, $|\Gamma|$ is the total number of runs, and M is the number of program entities), the space complexity is mainly dependent on the space needed to store the program spectra and result error, namely, $O(|\Gamma|\times M)$ and $O(|\Gamma|)$. The time complexity is determined based on the time needed to calculate a program entity's suspiciousness score and a fault context's suspiciousness score, namely, $O(|\Gamma|\times M)$ and $O(|\Gamma|\times M\times|\Gamma_f|)$.

4 Empirical study

We built a prototype tool called FCLFL to implement our approach. We used this tool to conduct an empirical study on seven benchmarks, which were obtained from the subject infrastructure repository (SIR) [12]. We present an empirical evaluation that analyzes the impact of FCLFL on different factors, and which compares our approach with four LFL methods. In particular, we seek answers to the following four research questions:

RQ1. Is FCLFL effective to further improve the fault absolute rank?

RQ2. How does choosing different scales of the fault context impact FCLFL?

RQ3. How does choosing different suspiciousness measurements impact FCLFL?

RQ4. Does FCLFL complement existing LFL approaches?

4.1 Experimental design

4.1.1 *Subject programs*

We used seven C programs, which include tcas, print_tokens, replace, schedule, schedule2, print_tokens2, and tot_inf as the subject programs. To determine whether a test case succeeds or fails, we created a fault-free version of each program in accordance with the program fault descriptions. The properties of each subject are shown in Table 3. The 2nd and 3rd columns show the number of lines of uncommented statements (LOC) and the basic blocks in these programs, respectively. The 4th column shows the number of faulty versions. The 5th column shows the number of test cases for each program version. The last column presents a description of each subject program. Each subject program contains one or more fault(s).

The Siemens suites have been widely used in many fault localization studies. There exist 132 versions in the suites, and there exist 10 versions that are seeded with bugs in variable declarations as follows: versions 13, 14, 15, 36, 38 for tcas; version 12 for replace; versions 6, 10, 19, 21 for tot_info. Because instrumentation cannot reach these declarations, we denote the directly infected block or an adjacent executable block as the fault location in order to reflect the effectiveness of a technique. We ignored print_tokens versions 4 and 6 because they are identical to the bug-free version. We excluded schedule2 version 9, as running all test cases only produces correct executions, namely, they have no failed test case. In total, the subjects of our experiment include 129 program versions of the Siemens suite from SIR [12].

4.1.2 *Improvement metric*

Effectiveness metrics are always required to obtain accurate and objective comparisons. There are mainly two metrics for measuring the fault-localization quality in a field.

• *PDG-based metric*. Let us suppose that a programmer locates bugs originating from suspicious program entities and performs a breadth-first search until he reaches one of the faults in the program-dependent graph (PDG). The metric is defined as T_score based on a static PDG. This metric was originally proposed by Renieris and Reiss [16], and was later adopted by many researchers [17,18]. The T_score can be defined as shown in (6):

$$T = \frac{|V_{\text{examined}}|}{|V|} \times 100\%, \tag{6}$$

where $|V|$ is the size of the PDG G and $|V_{\text{examined}}|$ is the size of the program entity set covered by the breadth-first search in G used to reach one of the defect-inducing bugs.

• *Ranking-based metric*. Tarantula, Jaccard, and Dstar et al. assumed that a programmer examines the program entities from the top of a ranking one-by-one until a fault is reached. The T_score is calculated directly from the ranking [19]. The ranking-based metric is defined as the percentage of program entities examined, as shown in (7):

$$T = \frac{|E_{\text{examined}}|}{|E|} \times 100\%, \tag{7}$$

where $|E|$ is the size of the ranking and $|E_{\text{examined}}|$ is defined as the number of examined program entities. The T_score can also be defined as $1 - T$. Based on this ranking-based metric, a metric value of 1 means that the faulty program entity is the first program entity in the ranked list and that there is an ideal situation.

We used the two standard effectiveness metrics for LFL mentioned above to normalize the rank of faulty program entities with respect to the size of the program. For instance, the root cause for a program A with 1200 blocks with a rank of 52, when expressed as a percentage, suggests that only 4.33% of the BBs will be checked. At first sight, this results may be considered fairly positive. However, in practice, this result may not help programmers in debugging [6,8,11]. Our goal is to improve the absolute rank; therefore, the above two metrics cannot be used directly to evaluate our approach. To measure the effectiveness of our approach, we focus on improving the absolute rank of suspicious program entities among the top-K-ranked program entities. Let B be the absolute rank of a root cause generated by an LFL tool for a buggy version, and A be the absolute rank of the root cause generated by our proposed approach for the same version; the improvement metric comparing the LFL with our approach can be simply defined, as shown in (8):

$$\text{Improvement}^{\text{LFL}}_{\text{FCLFL}}(A, B) = \frac{B - A}{B} \times 100\%. \tag{8}$$

In the metric, we assign the absolute rank's index starting from 0. Hence, we assume that the result of our approach is improved by 100% ($\frac{K-0}{K} \times 100\% = 100\%$) compared with that of an LFL if our approach ranks the root cause at the top of the ranking list. For example, if an LFL ranks the root cause in the top 20 for a buggy program, and our approach ranks it in the top 17, we can deduce that our improvement is $\frac{19-16}{19} \times 100\% = 15.8\%$.

4.2 Results

4.2.1 *RQ1: Is FCLFL effective to further improve fault absolute rank?*

The goal of this research question is to determine whether our approach is effective. We call the new method FCLFL-Dstar. To do this, we run Dstar and FCLFL-Dstar on the 129 fault versions.

We used Dstar as the reference, and we used the improvement metric as mentioned above to estimate the improvement caused by our approach. Figure 3 shows the results of the study. From the 129 fault versions, our approach shows improvements over the Dstar metric for 46 versions and performs the same as the Dstar metric for 83 versions, which indicates an effectiveness rate of 35.7%. The results show that no fault versions perform worse than the Dstar metric. Therefore, the fault-context approach is effective

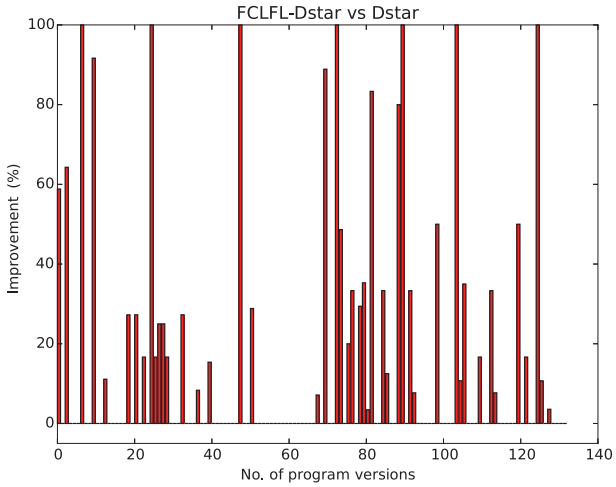

Figure 3 (Color online) Comparison of FCLFL-Dstar with Dstar.

Table 4 Improvement for different fault-context scales

Scale	(No. top K/top 1)$_{\text{Dstar}}$	Improvement number	Effective rate	Average improvement
K	53/12	11	26.8%	61.6%
$2K$	55/12	17	39.5%	64.0%
Top-all	129/12	20	–	63.5%

and can be used in practice. For those 46 versions, the average improvement is 65.18%, and it should be noted that in the fault-ranking report, the root causes of failures for seven buggy programs were ranked at the top.

Specifically, we investigated the effectiveness of our proposed approach when applied to multibug programs. Our dataset contains 11 faulty versions. We used FCLFL-Dstar to localize the multibugs. We assume that an LFL tool is effective if one of the multibugs appears within the top K. Therefore, we can also use metrics to evaluate the improvement caused by our approach. In the experiment, three buggy programs were improved; for example, one of the root causes of print_tokens v_1 is ranked from 13 to 5, while the remainder perform as well as Dstar. Therefore, we have reason to believe that our approach could be used in multibug settings.

4.2.2 *RQ2: How does choosing different scales of fault-context impact FCLFL?*

According to the definition of fault context, the fault context of program entity e_j is the set of all program suspicious entities, except for program entity j, that appear in the failed execution. Therefore, the number of program suspiciousness entities may influence our result. The goal of this study is to compare the effectiveness of FCLFL-Dstar using different scales of the fault context. Le et al. [11] reported that effective LFL tools should rank the root cause of failures within the top 10 rankings. Kochhar et al. [7] performed an empirical study, and their results showed that more than 80% of respondents indicated that they view a fault-localization session as successful only if it can localize bugs within the top five positions. In our experiment, we set K as $\min(10, 10\% \cdot \text{the number of BBs})$. To do this, we set the following different scales of the fault context: $\frac{1}{2}K$, K, $2K$, and size of (top-all). The value of K for each faulty version is shown in Table 3. For example, program print_tokens has 122 BBs, the value of K can be computed, and the result is $\min(10, 122 \cdot 10\%) = 10$.

Our result shows that our approach offers no improvement when we set the scale of the fault context to $\frac{1}{2}K$. Therefore, we ignore the result obtained for the fault-context scale $\frac{1}{2}K$, and give only the results of the other fault-context scales, which are shown in Table 4.

The results of our experiment are summarized in Table 4. We will not explain the column headers one-by-one, as they are self-explanatory. We only note that the second column represents the number of buggy programs for which the root cause of failures is ranked within the top X (X is K, $2K$, and size

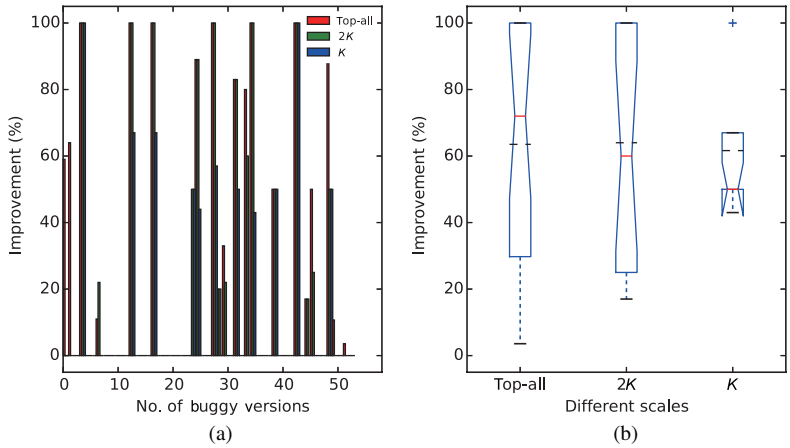

Figure 4 (Color online) Comparison of the effectiveness of different scales for the fault context. (a) Comparison effectiveness for different scales; (b) average improvement for different scales.

of (top-all)), and for which the root fault is ranked at the top of the ranking list. For example, there are 53 programs for which the root fault BB is ranked within the top K in the fault-ranking report generated for Dstar, and there are 12 programs for which the real fault BB is ranked at the top of the ranking list. Obviously, it is impossible for those 12 programs to improve; thus, we ignore them. As shown in Table 4, when the scale is $2K$, we can improve 17 programs out of 43 programs (55–12=43), which indicates an effectiveness rate of 39.5%, and the average improvement is 64.0% for those 17 programs. Similarly, when the scale is set as K and size of (top-all), the average improvement for those improved program versions is 61.6% and 63.5%, respectively.

As shown in Figure 4, the scale size of (top-all) is better than others, and the improvement has the same trend for most programs. The result also confirms that our approach can improve the absolute ranking further if an LFL can effectively locate some faults in certain programs.

4.2.3 *RQ3: How does choosing different suspiciousness measurements impact FCLFL?*

We also investigated whether our approach may be applied to other LFL methods aside from Dstar. We used the same 129 faulty versions and performed the same experiment to evaluate the effectiveness of combining our FCLFL approach with three other LFL tools: Jaccard, Tarantula, and Ochiai. In our experiment, we first ran an LFL approach to generate a suspiciousness rank list, and we obtained the number of buggy program versions for which the root cause of failures is ranked within the top K and as the top one. Then, we performed the same experiment using our FCLFL, which integrates the LFL suspiciousness metric.

The results are shown in Table 5, where $(\text{No. top } K/\text{top } 1)_{\text{LFL}}$ represents the number of buggy versions for which the root fault is ranked within top K/top 1 based on the LFL method, and $(\text{No. top } K/\text{top } 1)_{\text{FCLFL}}$ represents the number of buggy versions for which the root fault is ranked within top K/top 1 based on our FCLFL method. We observe that our proposed FCLFL approach can be improved when combined with either Tarantula, Jaccard, or Ochiai, similar to the case with Dstar. Let us look closer at the data. One observation is that although our approach improves the absolute rank compared with the LFL approach, the number of buggy program versions for which the root cause of program failures is ranked within the top K is not noticeably improved. For instance, the number of buggy program versions for which the real fault is ranked within the top K based on our approach combined with Dstar improved from 51 to 53. We believe that this is because those buggy programs that rank outside the range of the top K are not suitable for the use of the LFL approaches. Another observation is that our approach is effective for buggy programs for which the root cause of program failures is ranked within the top K. For instance, there are 19 buggy programs that are improved based on our approach combined with Ochiai, with the average improvement being 60.53% for those 19 buggy programs. It should be noted that the root causes of failure for seven buggy programs are ranked at the top of the ranking list. We confirm our

Table 5 Comparison with different suspiciousness metrics

LFL	(No. top K/top 1)$_{\text{LFL}}$	(No. top K/top 1)$_{\text{FCLFL}}$	Improvement number	Average improvement
Dstar	53/12	56/19	20	63.51%
Tarantul	52/12	54/17	19	60.36%
Jaccard	50/1	51/7	17	60.36%
Ochiai	55/14	58/20	19	60.53%

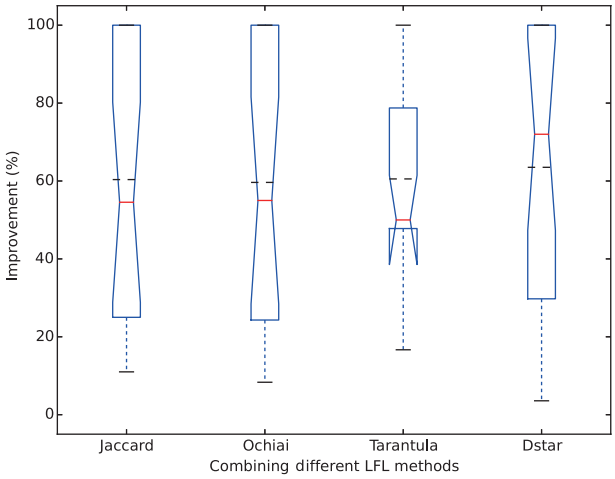

Figure 5 (Color online) Comparison of the effectiveness of different LFL methods for the fault context.

assumption that our proposed approach can be further improved if the root fault is ranked high in the ranking list.

Figure 5 shows the improvement based on different fault-localization metrics. We note that a similar effectiveness can be achieved when using different fault-localization metrics. Comparatively, FCLFL-Dstar is better than others.

4.2.4 *RQ4: Does FCLFL complement existing LFL approaches?*

FCLFL can be categorized as an LFL approach. Thus, we are interested in studying whether a direct correlation exists among the typical LFL approaches when computing fault absolute rankings. We can combine those four approaches to further improve the absolute rankings if the approaches do not have a direct correlation, and our approach does not complement existing LFL approaches in the case. We performed experiments to compare the four LFL approaches to verify whether they can further improve each other fault absolute ranking. We used 129 faulty versions and performed the same experiment to run the four LFL approaches and compare their fault absolute rank.

The results are shown in Figure 6, where D refers to Dstar, J refers to Jaccard, O refers to Ochiai, and T refers to Tarantula. Although there are a few exceptions, most root faults ranked higher in one approach while being higher in other approaches, vice versa. We can easily conclude that there exists a direct correlation for generating absolute fault rank among the four LFLs. The results illustrate two problems. (1) The ineffectiveness case in one LFL, and ineffectiveness in other cases. Therefore, we cannot study an effective suspiciousness measurement to improve those ineffectiveness cases, and we should ignore those cases that are applied in LFL approaches. (2) Simply combining LFL approaches cannot further improve the fault absolute rank even if those root faults are ranked higher. Therefore, FCLFL complements existing LFL approaches if an LFL approach is effective for a buggy program. We also plot the line chart that shows the top K components that need to be checked in order to find the root fault (x-axis) vs. the percentage of faults localized (y-axis), which is shown as Figure 7. Let us revisit the goal of FCLFL. We combine the suspiciousness of the program entity and the suspiciousness of its fault-context to further improve the absolute ranking for those root faults that have a higher rank. In those buggy programs, the contained root fault ranked lower, the suspiciousness of root fault is lower,

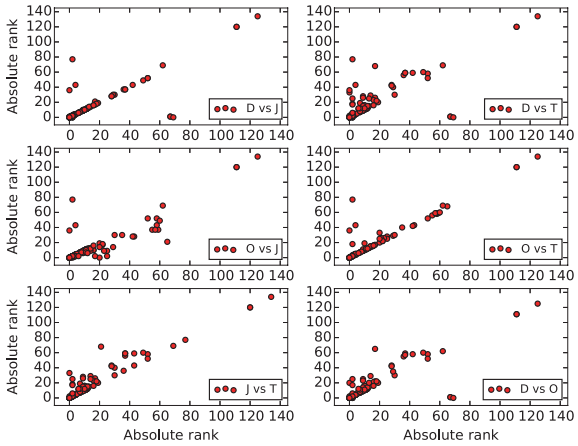

Figure 6 (Color online) Comparison of absolute rank between LFLs.

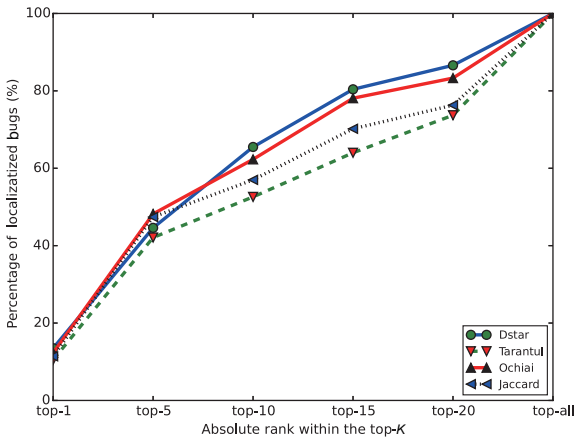

Figure 7 (Color online) Cumulative comparison for LFLs.

and we cannot obtain a satisfactory result even if the suspiciousness of the fault context is higher. Hence, our FCLFL only further improves those root faults that are ranked higher.

5 Discussion

5.1 Related works

There are many studies on LFL. Jones et al. [3] proposed Tarantula, which ranks program entities based on the set of failed and successful executions that cover the entities. Various other suspiciousness metrics have been proposed, such as [4,13,17,18]. Wong et al. [19] proposed an approach that applies radical basis function (RBF) networks for fault localization. In another study, Wong et al. [20] proposed a crosstab-based statistical approach for fault localization. There exist many variations of the above methods, which involve optimizing the test case suite [21–24] and using multiple coverage information [25, 26].

As previously mentioned, our goal is to further improve the absolute rank for buggy programs if LFL techniques can be effectively applied to those programs. In previous works, several LFL techniques have been combined with program analyses, and this approach could also be used to further improve the absolute rank. Zhang et al. [27] used edge profiles to represent successful executions and failed executions, and they contrasted them to model how each basic block contributes to failures by abstractly propagating the infected program states to their adjacent basic blocks through control flow edges. In their model, if a block is found to not transfer control to other successor blocks in the same CFG (as in the case of a return statement or callback function), it is referred to as an exit block. These exit blocks can be computed directly based on a specific metric. The authors then developed a set of linear algebraic equations based on the suspiciousness scores of basic blocks. Based on the suspiciousness of the exit blocks, the suspiciousness of all other basic blocks can be solved via standard mathematical techniques. Therefore, their approach can balance each suspiciousness score among infected blocks through an in-procedural control flow. Our fault context could also be viewed as a control flow context because we only considered program spectra. Therefore, our approach could also be used to balance each suspiciousness score among infected blocks that are propagated through a control flow. Note that as opposed to [27], our approach is lightweight, and may be used to balance the inter-procedural control flow.

Naish et al. [28] highlighted that failed test cases that cover fewer statements should be given more weight, and should have more influence on the fault rank and associated varying weights with the failed test cases. Their approach, which is similar to other LFL approaches, ignores the fault context. We can view the approach as a variation of Dstar, and the weights of failed test cases can cause each program entity to have different weights for n_{11}. Santelices et al. [26] proposed an LFL using multiple coverage

types. Their study shows that no single coverage type performs best for all faults. In other words, different kinds of faults are best localized based on different coverage types. Based on their results, they proposed a new LFL that leverages the unique qualities of each coverage type by combining them. Xu et al. [29] proposed an approach to improve the fault-localization technique based on branch-coverage spectra. Those LFL approaches can be combined with our proposed LFL approach to further improve the absolute rank.

Baah et al. [30] presented a linear estimation model for determining the causal effect of covering a given statement regarding the occurrence of failures, which is intended to eliminate or reduce confounding bias, and thereby yield a better fault-localization ranking by controlling the coverage of a statement's forward-control dependence predecessor. The statement's forward-control dependence predecessor can also be viewed as the statement's context for in-procedural control flow, and the context is generated based on the probabilistic program-dependence graph (PPDG), which is a heavyweight approach. Despite their application of Pearl's back-door criterion to the control-dependence graph of a program to justify the causal inference estimator, the bias triggered by the inter-procedural control flow could not be reduced.

The results of our empirical studies confirm that many root causes in buggy programs that are ranked higher by an LFL tool can have their absolute rank further improved by our approach. However, our studies also indicate that some root causes cannot be improved. To explain why those root causes cannot be improved, we checked and analyzed the relation between those root causes and program entities that are ranked higher. There are two reasons that could affect our approach. First, the root cause and those program entities that are ranked higher have the same execution path in their respective failed runs. Second, those programs that involve failed runs have much greater improvements than those of the root faults.

5.2 Threats to validity

The internal validity threat to our proposed approach mainly lies in the bias of experimenters. In our experiments, we collected program execution traces by running instrumented programs that are manually instrumented. Owing to the negligence of experimenters, there may be some bias, such as skipping or misplacing some blocks in instrumented programs. The main threat to the external validity of our finding is that our results may not be applicable to all programs. In our experiments, we used Siemens suites as experiment subjects that have been widely studied in past studies, and there are many test suites for each fault version. However, not many programs have a large number of practical test suites. Therefore, we should conduct more experiments for other programs to further reduce the threat. The threats to construct validity are mainly as follows: First, the threat lies in the suitability of our improvement metric. We adopted the improve_b^a as an improvement metric for the fault absolute rank. However, other improvement metrics of the fault absolute rank should be considered. Secondly, the threat lies in the suitability of the suspiciousness measurement. In order to reduce the threat, we compared four suspiciousness measurements, namely Tarantula, Jaccard, Ochiai, and Dstar. Finally, the threat lies in the definition of the effectiveness for LFL tools. In our study, we defined an LFL tool as being effective if one of the root faults is ranked at top K (K is defined as $\min(10, 10\% \times \text{len}(\text{program entities}))$ in their rank list). To reduce the threat, we considered different scales of K. However, those definitions are all based on the absolute rank. Other definitions for the effectiveness of LFL tools could also be considered, and we leave the consideration of other effectiveness definitions of LFL tools for future work.

6 Conclusion

In this study, our proposed approach involved the ranking of suspicious program entities based on the notions of suspiciousness of a program entity and the suspiciousness of its fault context. The higher the suspiciousness of a program entity, the lower the suspiciousness of its fault context, and the higher that the program entity will be ranked. Our experimental results show that our proposed approach is effective with respect to improving the fault absolute rank if LFL tools can be trusted.

There are two major directions in which we aim to continue our work. First, we plan to investigate more empirical studies to further evaluate the effectiveness of our approach. Second, this work is part of a greater effort to develop automatic debugging approaches. The next step in our project is to build a fault-context structure to aid in fault understanding. We believe that the fault context will help improve the fault absolute rank and to better understand root causes of failures.

Acknowledgements This work was supported by National High Technology Research and Development Program of China (863) (Grant No. 2015AA015303), National Natural Science Foundation of China (Grant Nos. 61272083, 61562087, 71371012, 61300170, 61572033), Key Support Program Projects for Outstanding Young Talents of Anhui Province (Grant No. gxyqZD2016124), Advanced Research of National Natural Science Foundation (Grant No. 2016yyzr10), and Anhui Natural Science Foundation (Grant Nos. KJ2016A252, 1608085MF147).

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Wong W E, Gao R, Li Y, et al. A survey on software fault localization. IEEE Trans Softw Eng, 2016, 42: 707–740
2 Chen M Y, Kiciman E, Fratkin E, et al. Pinpoint: problem determination in large, dynamic internet services. In: Proceedings of International Conference on Dependable Systems and Networks, Bethesda, 2002. 595–604
3 Jones J A, Harrold M J, Stasko J. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering, Orlando, 2002. 467–477
4 Abreu R, Zoeteweij P, Golsteijn R, et al. A practical evaluation of spectrum-based fault localization. J Syst Softw, 2009, 82: 1780–1792
5 Jones J A, Harrold M J. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. Long Beach: ACM, 2005. 273–282
6 Parnin C, Orso A. Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 International Symposium on Software Testing and Analysis, Toronto, 2011. 199–209
7 Kochhar P S, Xia X, Lo D, et al. Practitioners' expectations on automated fault localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, Saarbrcken, 2016. 165–176
8 Xie X, Liu Z, Song S, et al. Revisit of automatic debugging via human focus-tracking analysis. In: Proceedings of the 38th International Conference on Software Engineering, Austin, 2016. 808–819
9 Xia X, Bao L, Lo D, et al. "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. In: Proceedings of 2016 IEEE International Conference on Software Maintenance and Evolution. Raleigh: IEEE, 2016. 267–278
10 Thung F, Lo D, Jiang L. Are faults localizable? In: Proceedings of the 9th IEEE Working Conference on Mining Software Repositories, Zurich, 2012. 74–77
11 Le T D B, Lo D, Thung F. Should I follow this fault localization tool's output? Automated prediction of fault localization effectiveness. Empir Softw Eng, 2015, 20: 1237–1274
12 Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. Empir Softw Eng, 2005, 10: 405–435
13 Wong W E, Debroy V, Gao R Z, et al. The Dstar method for effective software fault localization. IEEE Trans Reliab, 2014, 63: 290–308
14 Harrold M J, Rothermel G, Sayre K, et al. An empirical investigation of the relationship between spectra differences and regression faults. Software Test Verif Reliab, 2000, 10: 171–194
15 Choi S S, Cha S H, Tappert C C. A survey of binary similarity and distance measures. J Syst Cybern Inf, 2010, 8: 43–48
16 Renieres M, Reiss S P. Fault localization with nearest neighbor queries. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Montreal, 2003. 30–39
17 Liblit B, Naik M, Zheng A X, et al. Scalable statistical bug isolation. ACM SIGPLAN Notices, 2005, 40: 15–26
18 Liu C, Yan X, Fei L, et al. SOBER: statistical model-based bug localization. ACM SIGSOFT Software Eng Notes, 2005, 30: 286–295
19 Wong W E, Debroy V, Golden R, et al. Effective software fault localization using an RBF neural network. IEEE Trans Reliab, 2012, 61: 149–169
20 Wong W E, Debroy V, Xu D X. Towards better fault localization: a crosstab-based statistical approach. IEEE Trans Syst Man Cybern Part C-Appl Rev, 2012, 42: 378–396
21 Yu Y, Jones J A, Harrold M J. An empirical study of the effects of test-suite reduction on fault localization. In: Proceedings of the 30th International Conference on Software Engineering. Leipzig: ACM, 2008. 201–210
22 Xia X, Gong L, Le T D B, et al. Diversity maximization speedup for localizing faults in single-fault and multi-fault programs. Autom Softw Eng, 2016, 23: 43–75

23 Xie X, Wong W E, Chen T Y, et al. Spectrum-based fault localization: testing oracles are no longer mandatory. In: Proceedings of the 11th International Conference on Quality Software, Madrid, 2010. 1–10
24 Gong D D, Wang T T, Su X H, et al. A test-suite reduction approach to improving fault-localization effectiveness. Comput Lang Syst Struct, 2013, 39: 95–108
25 Lo D, Xia X. Fusion fault localizers. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, Vasteras, 2014. 127–138
26 Santelices R, Jones J, Yu Y, et al. Lightweight fault-localization using multiple coverage types. In: Proceedings of IEEE 31st International Conference on Software Engineering, Vancouver, 2009. 56–66
27 Zhang Z, Chan W K, Tse T, et al. Capturing propagation of infected program states. In: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, Amsterdam, 2009. 43–52
28 Naish L, Lee H J, Ramamohanarao K. Spectral debugging with weights and incremental ranking. In: Proceedings of the 16th Asia-Pacific Software Engineering Conference. Batu Ferringhi: IEEE, 2009. 168–175
29 Xu S, Xu J, Yang H, et al. An improvement to fault localization technique based on branch-coverage spectra. In: Proceedings of the 39th IEEE Annual Computer Software and Applications Conference. Taichung: IEEE, 2015. 2: 282–287
30 Baah G K, Podgurski A, Harrold M J. Causal inference for statistical fault localization. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, Trento, 2010. 73–84