International Journal of Software Engineering
and Knowledge Engineering
Vol. 23, No. 7 (2013) 963–998
© World Scientific Publishing Company
DOI: 10.1142/S0218194013500289

A FAULT LOCALIZATION FRAMEWORK TO ALLEVIATE THE IMPACT OF EXECUTION SIMILARITY[*]

LEI ZHAO
*Key Laboratory of Aerospace Information Security
and Trust Computing, Wuhan 430072, P. R. China
and
Computer School of Wuhan University, Wuhan 430072, P. R. China
and
State Key Laboratory for Novel Software Technology
Nanjing University, Nanjing, P. R. China
zhaolei.whu@gmail.com*

ZHENYU ZHANG
*State Key Laboratory of Computer Science, Institute of Software
Chinese Academy of Sciences, Beijing 100190, P. R. China
zhangzy@ios.ac.cn*

LINA WANG
*Key Laboratory of Aerospace Information Security and
Trust Computing, Wuhan 430072, P. R. China
and
Computer School of Wuhan University, Wuhan 430072, P. R. China
lnwang@whu.edu.cn*

XIAODAN YIN
*Computer School of Wuhan University, Wuhan 430072, P. R. China
yinxiaodan.whu@gmail.com*

Received 13 June 2012
Revised 24 September 2012
Accepted 25 March 2013

Coverage-based fault localization (CBFL) techniques contrast the execution spectra of a program entity to assess the extent of how much a program entity is being related to faults. However, different test cases may result in similar executions, which further make the execution spectra of program entities be indistinguishable among similar executions. As a consequence, most of the current CBFL techniques are impacted by the noise of indistinguishable spectra. To alleviate the impact of execution similarity and improve the effectiveness of CBFL techniques, we propose a general fault localization framework. This framework is general to current

[*]A preliminary version of this paper was published in the *23rd International Conference on Software Engineering and Knowledge Engineering* (*SEKE* 2011).

execution spectra based CBFL techniques, which could synthesize a fault localization technique based on a given base technique. To synthesize the new technique, we use the concept of coverage vector to model execution spectra and capture the execution similarity, then reduce the impact of execution similarity by counting distinct coverage vectors, and finally assess the suspiciousness of basic blocks being related to faults with the spectra of distinct coverage vectors. We adopt four representative fault localization techniques as base techniques, use seven Siemens programs and three median-sized real-life UNIX utility programs as subject programs, to conduct an experimental study on the effectiveness of our framework. The empirical evaluation shows that our framework can effectively alleviate the impact of execution similarity and generate more effective fault localization techniques based on existing ones.

Keywords: Fault localization; coverage vector; execution similarity.

1. Introduction

Software has been extensively used in the government sectors, research institutes, commercial companies, and so on. Not only such organizations but also our daily life fundamentally relies on software. However, software defects are still hard to be completely avoided in software releases. As a result, software failures caused by defects lead to the loss of private properties, huge costs to business plans, and even serious disasters. How to improve the quality of software and reduce the chance of software failures is always a realistic but challenging task in software engineering research.

Many software failures are caused by program faults [1], which are mistakenly imported into software by developers. To detect and fix such program faults, software debugging is a significant and practical process [2–4]. Typically, a debugging task involves at least three steps [5, 6]: (1) locate the faults, (2) fix the located faults, (3) and regressively test the repaired program. During this process, fault localization has been recognized as the most difficult, tedious, and time-consuming step [7, 8]. Using an automatic and effective technique to assist fault localization is promising to alleviate the problem.

Many kinds of fault localization techniques have been proposed in the past decades. For example, by isolating failure-inducing input components and analyzing program state changes during a failed execution against a successful one, delta debugging [9–11] produces cause-effect chains and locates suspicious statements. Dynamic slicing based fault localization techniques narrow down the suspicious statements using backward slicing approach [12]. Gupta *et al.* [13] combine the delta debugging, use both the forward and backward slicing to narrow down statements. However, delta debugging and dynamic slicing may not be effective for large scale software due to the complicated state propagation [14].

Coverage-based fault localization (CBFL) techniques [15, 16, 3, 17, 18] are a popular family of fault localization techniques. Previous studies showed that CBFL techniques are effective in locating faults [8]. The key insight behind CBFL is that dynamic features (such as execution spectra) of fault-relevant program entities (such as statements, blocks, and predicates) are more sensitive to the differences between the set of failed executions and the set of passed executions [15, 16, 3, 19]. By applying a statistical approach to correlate program entities with failures, and formulating a comparison function to contrast the dynamic features of program entities between passed and failed executions, CBFL techniques can locate the program entities, which are strongly correlated to the observed execution failures.

There are several representative CBFL techniques, such as Tarantula [15, 8], CBI [3, 20], SBI [21], Ochiai [22], Jaccard [22], and so on. For each statement, Tarantula [15, 8] calculates the ratio of failed executions exercising the statement and the ratio of passed executions exercising it, and uses the ratio of the former to the sum of the former and the latter to assess the extent of how much that statement is related to faults. CBI [3, 20] compares the probability that a program fails when a predicate is ever evaluated to be true with the probability that the program fails when the predicate is ever evaluated, and use the former subtracted from the latter to assesses the possibility of the predicate being related to faults. Yu *et al.* [21] modify CBI to a statement-level technique SBI. For each statement, SBI calculates the number of failed executions exercising the statement and the number of passed executions exercising it, and uses the ratio of the former to the sum of the former and latter to assess the extent of how much each statement being related to faults. Abreu *et al.* [22] adopt the Ochiai and Jaccard coefficient factors to design the comparison formula and evaluate the suspiciousness scores of statements. There are some other similar techniques [23, 24], which employ different heuristics to assess the suspiciousness of a program entity being related to faults and sort the program entities into a list in the descending order of the calculated suspiciousness scores.

During software testing, test cases should be designed to cover different branches or paths, so as to manifest more program states [25, 26]. However in practice, it is common that test cases may not always be generated to satisfy some coverage criteria, and there is no guarantee that the test suite reduction task is always conducted before testing to eliminate similar executions. As a result, different executions still have chance to generate similar or even identical execution spectra [21, 27]. In previous CBFL techniques, most of them ignore the execution similarities, but such similar or even identical execution spectra can be indistinguishable, in terms of program coverage, which may bring negative impact on the effectiveness of fault localization techniques [6, 48, 28]. For example, Hao *et al.* [6] show that the similarity may harm the effectiveness of CBFL techniques. Wong *et al.* [48] demonstrate the contribution of different test cases for CBFL techniques. It is suggested that for a CBFL technique, similar execution spectra should not have as much contribution to the effectiveness of fault localization as others. On the other hand, when a passed execution and a failed execution share the same execution path, coincidental correctness may have occurred [28]. In coincidental correctness cases, the fault-relevant program entity is exercised in both passed and failed executions, and thus becomes difficult to be recognized via contrasting the corresponding execution spectra. In one word, the execution similarity could have negative impact on the effectiveness of previous fault localization techniques or even make them lose effect. Previous study also shows that such execution similarity is a frequent phenomenon in realistic [29].

In our previous work [30], we have proposed a novel CBFL approach to measure execution similarity, an approach to eliminate test cases associating with similar execution, and a fault localization technique with similarity reduction. However, our previous approach is limited by a specific comparison function to assess the fault related program entities. Since most previous CBFL techniques shall be impacted by execution similarity, we are interested in the problems that how to capture the similarity and whether previous CBFL techniques could be improved.

In this paper, we propose a general framework named PAFL to address the problem that execution similarity may reduce the effectiveness of fault localization. In our framework, we mainly focus on two research questions: (1) how to evaluate the execution similarity, and (2) how to alleviate the impact of execution similarity on fault localization. Our framework is designed to synthesize a new fault localization technique from a base one. In our framework, we first use the concept of coverage vector to count distinct execution paths and capture the presence of execution similarity. Then based on the base technique, we calculate the failing extent of each distinct coverage vector, and mark it as a failed or passed distinct coverage vector according to whether it is ever exercised in a failed execution or not. Next, for each basic block, we evaluate the suspiciousness score of that basic block using the execution statistics of distinct coverage vectors. At last, we sort all the basic blocks in the descending order of their computed suspiciousness scores.

We select four representative techniques, Ochiai [22], Jaccard [22], Tarantula [15], and SBI [21], as base techniques, use seven Siemens programs and three median-sized real-life UNIX utility programs to evaluate our framework. The empirical evaluation results show that our framework can synthesize a fault localization technique that is more effective than the base one, thus alleviates the impact of execution similarity on the selected subject techniques and programs. We further compare the effectiveness of the best synthesized technique in our framework with existing related work SAFL [6] and ICST10 [31], which also have the concept of eliminating similar executions, and find that the best technique synthesized in our framework is more promising than SAFL and ICST10 in locating faults, in the case of the presence of execution similarity.

The contributions of this paper are two-fold. (1) We propose a general framework to synthesize new fault localization techniques from base ones, in order to address the problem of execution similarity; (2) We adopt four representative fault localization techniques as base techniques, use seven Siemens programs and three median-sized real-life UNIX utility programs as subject programs, to conduct an experimental study on the effectiveness of our framework. The empirical evaluation shows that our framework can alleviate the impact of execution similarity, and synthesize more effective fault localization techniques based on existing ones.

We organize the rest of this paper as follows. Section 2 uses a concrete example to demonstrate previous techniques and motivate our idea. Section 3 elaborates on our framework and illustrates it using the example in Sec. 2. Section 4 presents the controlled experiment and analyzes the results. Sections 5 and 6 review related work and conclude the paper, respectively.

2. Motivation

In this section, we use an example to demonstrate previous techniques and motivate our idea.

2.1. *Motivating example*

The code excerpt in Fig. 1 finds the middle value in three given numbers. A fault exists in statement s_{12}, which mistakenly accesses the variable x instead of m. As an unexpected result, the program may output incorrect results because the program

Blocks		Statements	Test cases									Previous techniques							
			t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	Ochiai		Jaccard		Tarantula		SBI	
			(1,2,3)	(3,1,2)	(3,2,3)	(1,3,2)	(1,3,3)	(1,1,1)	(2,3,2)	(2,1,3)	(3,4,1)								
			F	***F***	*P*	***F***	*P*	*P*	*P*	*P*	*P*	score	rank	score	rank	score	rank	score	rank
***b*₁**	 **s₁** **s₂** **s₃**	`Mid() {` `int x, y, z, m;` `read ("Enter:", x, y, z);` `m=z;` `if (y<z){`	•	•	•	•	•	•	•	•	•	0.58	4	0.33	4	0.5	5	0.33	5
***b*₂**	**s₄**	`if (x<y)`	•	•	•					•		0.58	4	0.4	1	0.67	2	0.5	2
***b*₃**	**s₅**	`m=y;`	•									0.58	4	0.33	4	1.0	1	1.0	1
***b*₄**	**s₆**	`else if (x<z)`		•	•					•		0.33	5	0.2	5	0.5	5	0.33	5
***b*₅**	**s₇**	`m=x;` `}`								•		0.0	9	0.0	9	0.0	9	0.0	9
***b*₆**	**s₈**	`else if (x>y)`				•	•	•	•		•	0.26	7	0.14	7	0.33	7	0.2	7
***b*₇**	**s₉**	`m=y;`																	
***b*₈**	**s₁₀**	`else if (x>z)`				•	•	•	•		•	0.26	7	0.14	7	0.33	7	0.2	7
***b*₉**	**s₁₁**	`m=x;`									•	0.0	9	0.0	9	0.0	9	0.0	9
***b*₁₀**	**s₁₂**	`printf("The middle is:", x);` **`/* a mutant of m */`** `}`	•	•	•	•	•	•	•	•	•	0.58	4	0.33	4	0.5	5	0.33	5
		Code examining effort to locate fault:										40%		40%		50%		50%	

Blocks	Distinct paths					Our approaches							
	p_1	p_2	p_3	p_4	p_5	on Ochiai		on Jaccard		on Tarantula		on SBI	
	t_1	t_2 t_3	t_4 t_5 t_6 t_7	t_8	t_9								
	F	***F***	***F***	*P*	*P*	score	rank	score	rank	score	rank	score	rank
***b*₁**	•	•	•	•	•	0.77	2	0.6	2	0.5	4	0.6	4
***b*₂**	•	•		•		0.67	3	0.5	3	0.57	2	0.67	2
***b*₃**	•					0.58	4	0.33	4	1.0	1	1.0	1
***b*₄**		•		•		0.41	7	0.25	7	0.4	7	0.5	7
***b*₅**				•		0.0	9	0.0	9	0.0	9	0.0	9
***b*₆**			•		•	0.41	7	0.25	7	0.4	7	0.5	7
***b*₇**													
***b*₈**			•		•	0.41	7	0.25	7	0.4	7	0.5	7
***b*₉**					•	0.0	9	0.0	9	0.0	9	0.0	9
***b*₁₀**	•	•	•	•	•	0.77	2	0.6	2	0.5	4	0.6	4
Code examining effort to locate fault:						20%		20%		40%		40%	

Fig. 1. The motivating example and execution spectra.

should output the value of m instead of x. We know that the faulty statement (s_{12}) and the faulty basic block (b_{10}) will be exercised in every run. However, when we look at the execution results of nine test cases, we find that this program can still output correct results for six among the nine test cases. Apparently, coincidental correctness [28, 31, 29] happens frequently in this program. When coincidental correctness occurs, it is difficult for programmers to recognize a passed program execution [31, 29], which exercises the faults.

There are 9 test cases in Fig. 1, named t_1 to t_9. Each test case refers to a triple of 3 numbers, and these 3 numbers are the values of x, y, z, respectively. For the 9 test cases, we use the mark "•" in the cell to indicate that a statement is exercised in the program execution with respect to a test case. The execution results (P for pass and F for fail) are shown in the table header. The execution spectra for each basic block can be calculated through the marked "•" and execution results. For example, b_1 is exercised in each of the three failed executions and each of the six passed executions. We use $\mathit{failed}(b_1) = 3$ and $\mathit{passed}(b_1) = 6$ to denote these two numbers, respectively.

2.2. *Previous techniques*

We apply four previous techniques Ochiai [22], Jaccard [22], Tarantula [15], and SBI [21] to locate the fault (statement s_{12} or basic block b_{10}) in this example. All these four techniques are representative CBFL techniques, which are always employed as subject techniques in Sec. 4 of this paper.

The suspiciousness scores and rankings of blocks are also shown in Fig. 1. Let us take b_1 and the technique Tarantula for illustration. The formula of Tarantula [15] is $\frac{\%\mathit{failed}(b_1)}{\%\mathit{failed}(b_1)+\%\mathit{passed}(b_1)}$, where $\%\mathit{failed}(b_1)$ is the ratio of failed executions that exercise b_1 to the total number of failed executions, and $\%\mathit{passed}(b_1)$ is the ratio of passed executions that exercise b_1 to the total number of passed executions. In the example, the total numbers of failed executions and passed executions are 3 and 6, respectively. Therefore, $\%\mathit{failed}(b_1) = 1$ and $\%\mathit{passed}(b_1) = 1$. As a result, Tarantula assigns the suspiciousness score 0.5 to b_1. Larger suspiciousness score means that the block is more suspicious to contain faults.

In previous studies, programmers are suggested to search for faults by examining the basic blocks in descending order of their suspiciousness scores. The percentage of basic blocks examined before reaching a fault is regarded as the code examining effort, known as the *expense* evaluation metrics, to locate the fault using that technique. In Fig. 1, the order of a block to be examined is also given, named as rank. Take the result of Tarantula for illustration, b_3 has the highest suspiciousness score and thus be ranked as no. 1. For the faulty block, $b_1 0$, the suspiciousness score is no larger than other 5 blocks, and thus $b_1 0$ is ranked as no. 5. With such an evaluation metric, programmers finally need to examine 50% of all code to locate the fault, when applying Tarantula. The results of Ochiai, Jaccard, and SBI can be similarly explained.

Unfortunately, the results of these four techniques are not as good as expected and the code examining efforts to locate the fault are huge (e.g, 50%). By checking the computation of these techniques, we have the following findings. For example, Tarantula and SBI give b_3 the highest suspiciousness score, and this makes Tarantula and SBI somehow ineffective to locate the fault. The reason that b_3 gets the highest suspiciousness score is that it is only exercised in one failed execution and none passed execution. Tarantula and SBI calculate the suspiciousness score 1 to such trivial execution spectra. On the other hand, Ochiai and Jaccard have addressed such a common problem by decreasing the suspiciousness scores for basic blocks having such execution spectra. As a result, b_3 is not assigned the highest suspiciousness score by Ochiai and Jaccard, which finally need less code examining effort to locate the fault. Since the task of this example is not to explain how Ochiai and Jaccard manage to do that, let us continue our investigation on how to improve the effectiveness of the four techniques.

2.3. *Questioning the reasons*

The faulty statement s_{12} or basic block b_{10} happens to be exercised in all passed and failed executions, so it is hard to assess its suspiciousness being related to faults from execution spectra. On the other hand, b_4 is given higher suspiciousness score than b_6 and b_8 because the latter happens to be exercised in relatively more passed executions than the former. In details, b_4 is exercised in one failed execution (t_2) and two passed executions (t_3 and t_8), while b_6 and b_8 are exercised in one failed execution (t_4) and four passed executions (t_5, t_6, t_7, and t_9). Among the three test cases (t_2, t_3, and t_8) which exercise b_4, 33% of them are failed, while only 20% of the test cases that exercise b_6 and b_8 are failed. According to the basic insight of CBFL, executions exercising b_4 are more likely to fail. Therefore, b_4 is deemed more suspicious than b_6 and b_8.

In addition, we also observe that coincidental correctness may have occurred in the program executions of test cases t_3, t_5, t_6, and t_7, because t_2 and t_4 are failed test cases while they have the same execution path with t_3, t_6, t_7 and t_8. In this example, it seems that there are some clues to identify the occurring of coincidental correctness.

When looking at the executions for test cases t_4, t_5, t_6, and t_7, we find that they have identical coverage in terms of program execution. It is not strange because of the existence of coincidental correctness issue. To escape from the complicated reason of coincidental correctness cases, let we use the term "execution similarity" to name such a phenomenon. That is, when some test cases generate identical program execution, we observe similar execution and name such a phenomenon as "execution similarity". Previous studies have shown that execution similarity can be frequently observed in realistic programs [6, 27, 21]. To alleviate the impact of execution similarity, we use the execution spectra of paths to measure the execution similarity.

2.4. *Our approaches*

Since execution similarity is observed on t_4, t_5, t_6, and t_7 (t_4 finally reveals a failure), we know that it is not safe to mark t_5, t_6, and t_7 as passed test cases to continue computation. It is because that we now have evidence (from t_4) that the faulty statement must have been exercised in the program execution of t_5, t_6, and t_7. To play safe, we adopt a conservative strategy. We count the distinct coverage among executions for all test cases, and mark a distinct coverage as a "failed coverage" if it is ever associated with a failed test case as shown in Fig. 1. On the contrary, we mark a distinct coverage as a "passed coverage" if it is never associated with a failed test case. To ease our presentation, we also use the intuitive name "distinct path" to refer to such a "distinct coverage". The distinct paths are shown in the lower part of Fig. 1. By such marking, we know that distinct paths p_1, p_2, and p_3 are failed ones, distinct paths p_4 and p_5 are passed ones.

By such a transformation, we believe that we have alleviated the impact of coincidental correctness since we conservatively mark each failure-causing distinct path as failed. We use such distinct path as program unit, and apply a base CBFL technique to calculate suspiciousness for them. For example, we adopt the formula of Jaccard $\frac{\mathit{failed}'(b_i)}{\mathit{passed}'(b_i)+(\#\mathit{total\ of\ failed\ distinct\ paths})}$ to estimate the suspiciousness of a basic block being related to faults, where $\mathit{failed}'(b_i)$ stands for the number of failed distinct paths that exercise b_i, and $\mathit{passed}'(b_i)$ stands for the number of passed distinct paths that exercise b_i. For example, for b_1, we calculate its suspiciousness score as $\frac{3}{2+3} = 0.6$. Finally, we find that we need only 20% code examining effort to locate the fault.

Further, in the motivating example, it seems that we have found a way to generalize the idea to apply other kind of techniques. This time, suppose we adopt the formula of Ochiai $\frac{\mathit{failed}'(b_i)}{\sqrt{(\mathit{failed}'(b_i)+\mathit{passed}'(b_i))(\#\mathit{total\ of\ failed\ distinct\ paths})}}$ to estimate the suspiciousness of a basic block, where $\mathit{failed}'(b_i)$ stands for the number of failed distinct paths that exercising b_i, and $\mathit{passed}'(b_i)$ stands for the number of passed distinct paths that exercising b_i. Take b_1 for illustration again, its suspiciousness score is calculated $\frac{3}{\sqrt{5\times 3}} = 0.77$. Finally, we find that we need only 20% code examining effort to locate the fault. Again, the effectiveness of Ochiai is also improved. Even when we move to Tarantula and SBI, we find that with such a transformation, their effectiveness can be improved.

2.5. *Challenges*

The above example has interestingly demonstrated that previous techniques may not be effective when coincidental correctness happens, while our approach has the potential to partially address the problem. However, we also foresee some challenges. For example, how to formally define a "failed distinct path"? Is there any other technique on which our approach can be applied? Can our approach be generalized and apply on different CBFL techniques? Will our approach also work well on different CBFL techniques? In the next sections, we will elaborate on our model and answer these questions.

3. Our Framework

In this section, we give the problem settings, illustrate the whole picture of our framework, give preliminaries and definitions, and elaborate on our fault localization framework, which is named PAFL.

3.1. *Problem settings*

Let P be a program containing faults. We use $P = \{b_1, b_2, \ldots, b_n\}$ to denote that program P has n basic blocks (b_1 to b_n). We further use $T = \{t_1, t_2, \ldots\}$ to denote the set of test cases.

We further use the term $P(t_i)$ to denote the execution of P over t_i. We use the term $\mathit{failed}(t_i)$ to denote the failed execution of P over t_i, and the term $\mathit{passed}(t_i)$ to denote the passed execution of P over t_i. Our aim is to estimate the extent of how much each basic block b_i of P is related to faults. In this paper, we also use the term *suspiciousness score* of a basic block to denote such a value. We sort the basic blocks into a list in descending order of their calculated suspiciousness scores. Programmers may search along such a list to locate faults in programs [15, 4].

In Fig. 1, $P = \{b_1, b_2, \ldots, b_{10}\}$ is the Mid method. $T = \{t_1, t_2, \ldots, t_9\}$ is the set of the nine test cases.

3.2. *Preliminaries and definitions*

To ease our presentation, we involve in the definition of "coverage vector" to formally describe the concept of "distinct path" used in Sec. 2.

Definition 1. An **original coverage vector** $ocv_i = \langle b_1, b_2, \ldots, b_n\rangle (b_j \in \{0, 1\}$ for $j = 1, 2, \ldots, n)$ of program execution $P(t_i)$ (P's execution over t_i) is a tuple.

We use $ocv_i(b_j)$ to retrieve the *jth* element in the tuple, where $ocv_i(b_j) = 1$ means the basic block b_j is exercised in the execution, $ocv_i(b_j) = 0$ means b_i is not exercised in the execution. For the coverage vector ocv_i with respect to execution $P(t_k)$, we also say $P(t_k)$ **covers** ocv_i, and denote it as $ocv_i \in P(t_k)$.

In Fig. 1, there are nine original coverage vectors. The coverage vector with respect to test case t_1 is $ocv_1 = \langle 1, 1, 1, 0, 0, 0, 0, 0, 0, 1\rangle$.

Apparently, different executions (even both passed executions and failed executions) may have identical original coverage vectors. So let us move to the next definition.

Definition 2. The set of **distinct coverage vector set** $CV = \{cv_1, cv_2, \ldots\}$ is the distinct set (with no repeating elements) of all original coverage vectors ocv_i with respect to the program execution $P(t_k)$ of each test case t_k. Each element $cv_i \in CV$ is called a coverage vector. Similarly, we use $cv_i(b_j)$ to retrieve the *jth* element in the tuple of cv_i.

By such definition, we know that we have $cv_i \neq cv_j$ for any two coverage vectors cv_i and $cv_j (1 \leq i < j)$. In Fig. 1, there are five distinct coverage vectors, namely, $cv_1 = \langle 1,1,1,0,0,0,0,0,0,1 \rangle$, $cv_2 = \langle 1,1,0,1,0,0,0,0,0,1 \rangle$, $cv_3 = \langle 1,0,0,0,0,1,0,1,0,1 \rangle$, $cv_4 = \langle 1,1,0,1,1,0,0,0,0,1 \rangle$, and $cv_5 = \langle 1,0,0,0,0,1,0,1,1,1 \rangle$.

3.3. *A whole picture of our framework*

We recall that our aim is to estimate the extent of how much each basic block is related to faults, considering the impact of execution similarity. The main steps include: (1) how to estimate the execution similarity, and (2) how to evaluate the suspiciousness scores of basic blocks in case of presence of execution similarity. Our approach mainly addresses the two problems and a whole picture of our framework is shown in Fig. 2.

As shown in Fig. 2, when some executions reveal failures, we know that there must exist some faults in the program. The execution statistics are inputs of our fault localization framework. To capture the execution similarity, we make use of the coverage vector as a kind of program entity, and calculate its execution spectra. Executions with identical coverage vector are deemed similar. After that, we calculate the failing extent for each distinct coverage vector. For each distinct coverage vector, if there is a failed execution exercising it, that indicates that executions covering it ever reveal failures. If all executions exercising this coverage vector are passed ones, it indicates that among all executions covering this coverage vector, no one reveals failure. As demonstrated in the motivation section, we mark the coverage vectors which are ever exercised in a failed execution as *failed coverage vectors*, and mark the coverage vectors which are never exercised in a failed executions as *passed coverage vectors*. For each block, we next estimate the suspiciousness score by contrasting the execution spectra of the coverage vectors associated with it. Finally,

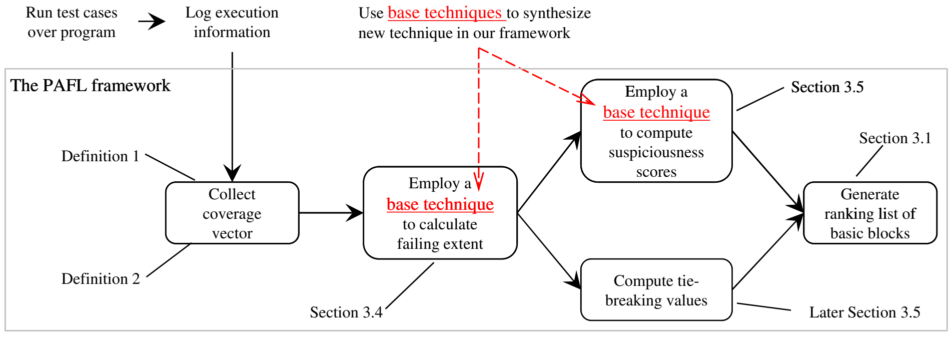

Run test cases over program
Log execution information
Use base techniques to synthesize new technique in our framework
The PAFL framework
Employ a base technique to compute suspiciousness scores
Section 3.5
Section 3.1
Definition 1
Collect coverage vector
Employ a base technique to calculate failing extent
Generate ranking list of basic blocks
Definition 2
Compute tie-breaking values
Section 3.4
Later Section 3.5

Fig. 2. A whole picture of our framework.

we generate a ranked list of basic blocks in descending order of their calculated suspiciousness scores. For tie-cases, we also have a tie-breaking strategy.

In our framework, we work on base techniques to synthesize new fault localization techniques. Compared with other fault localization frameworks (such as [32]), our framework is different in that it is designed to alleviate the impact of execution similarity. Compared with other CBFL techniques, the techniques synthesized in our framework calculate suspiciousness scores of blocks according to the execution spectra of coverage vectors, rather than statements or blocks. In addition, we do not limit the technique employed to calculate the failing extent and suspiciousness scores. Previous studies [18] introduce a general formula

$$S_T(b_i) = f_T(a_{np}(b_i), a_{nf}(b_i), a_{ep}(b_i), a_{ef}(b_i))$$

to represent the calculation of suspiciousness scores, where the four parameter $a_{np}(b_i)$, $a_{nf}(b_i)$, $a_{ep}(b_i)$, and $a_{ef}(b_i)$ stand for the number of passed executions that do not exercise b_i, the number of failed executions that do not exercise b_i, the number of passed executions that exercise b_i, and the number of failed executions that exercise b_i, respectively. They form a general problem setting for CBFL techniques, and 33 existing techniques have been expressed using such a setting [18, 19]. Since we also based on such a setting to propose our framework, our approach is a general framework for fault localization, in which many popular techniques can be applied.

In our framework, the computation include two steps, the calculation of failing extent and suspiciousness scores.

To ease the presentation, we use

$$F_T = f_T(a_{np}(t_j, cv_i), a_{nf}(t_j, cv_i), a_{ep}(t_j, cv_i), a_{ef}(t_j, cv_i))$$

to stand for the core formula of failing extent using a base fault localization technique T, where $a_{np}(t_j, cv_i)$, $a_{nf}(t_j, cv_i)$, $a_{ep}(t_j, cv_i)$, and $a_{ef}(t_j, cv_i)$ are four parameters about the execution statistics of the coverage vector cv_i. In this formula, $a_{np}(t_j, cv_i)$ stands for the number of passed executions that do not exercise cv_i, $a_{nf}(t_j, cv_i)$ stands for the number of failed executions that do not exercise cv_i, $a_{ep}(t_j, cv_i)$ stands for the number of passed executions exercising cv_i, $a_{ef}(t_j, cv_i)$ stands for the number of failed executions exercising cv_i, and F_T calculates the failing extent for the coverage vector cv_i.

We use

$$S_T = f_T(a_{np}(cv_j, b_i), a_{nf}(cv_j, b_i), a_{ep}(cv_j, b_i), a_{ef}(cv_j, b_i))$$

to stand for the core formula of suspiciousness scores when applying a base fault localization technique T, where $a_{np}(cv_j, b_i)$, $a_{nf}(cv_j, b_i)$, $a_{ep}(cv_j, b_i)$, and $a_{ef}(cv_j, b_i)$ are four parameters about the execution spectra of basic block b_i. In this formula, $a_{np}(cv_j, b_i)$ stands for the number of passed coverage vectors that do not cover b_i, $a_{nf}(cv_j, b_i)$ stands for the number of failed coverage vectors that do not cover b_i, $a_{ep}(cv_j, b_i)$ stands for the number of passed executions covering b_i, $a_{ef}(cv_j, b_i)$ stands for the number of failed executions covering b_i, and S_T calculates the suspiciousness score for the block b_i.

The general method to apply a base technique in our framework are introduced in the next two sections, which elaborate on F_T and S_T.

3.4. *Calculation of failing extent*

In our framework, we use the failing extent of a coverage vector to estimate the execution similarity.

There may be several coverage vectors that are covered by different failed executions, and a coverage vector may be covered by both passed executions and failed executions. For example, in Sec. 2, the coverage vector $cv_3 = \langle 1, 0, 0, 0, 0, 1, 0, 1, 0, 1\rangle$ is covered by one failed execution (t_4) and three passed executions (t_5, t_6, and t_7). We use the term "failing extent of a coverage vector" to capture the extent of an execution (which covers a specific coverage vector) failing. In the following, the failing extent of a coverage vector is denoted as $F(cv_i)$.

When employing a base technique to synthesize a fault localization technique in our framework, we adapt $F_T(cv_i)$ to calculate the failing extent of a coverage vector. The failing extent of a coverage vector cv_i is calculated as Eq. (1).

$$F_T(cv_i) = f_T(a_{np}(t_j, cv_i), a_{nf}(t_j, cv_i), a_{ep}(t_j, cv_i), a_{ef}(t_j, cv_i)). \tag{1}$$

In Eq. (1), f_T is the formula of the base technique T, $a_{np}(t_j, cv_i)$ stands for the number of passed executions that do not exercise cv_i, $a_{nf}(t_j, cv_i)$ stands for the number of failed executions that do not exercise cv_i, $a_{ep}(t_j, cv_i)$ stands for the number of passed executions exercising cv_i, and $a_{ef}(t_j, cv_i)$ stands for the number of failed executions exercising cv_i.

For example, if we use the equation of SBI [21] as the base technique, the failing extent is calculated using the formula of SBI $T_{\mathrm{SBI}}(b_i) = \frac{a_{ef}(t_j,b_i)}{a_{ef}(t_j,b_i)+a_{ep}(t_j,b_i)}$ as Eq. (2).

$$F_{\mathrm{SBI}}(cv_i) = \frac{a_{ef}(t_j, cv_i)}{a_{ef}(t_j, cv_i) + a_{ep}(t_j, cv_i)}. \tag{2}$$

In Eq. (2), $a_{ef}(t_j, cv_i)$ and $a_{ep}(t_j, cv_i)$ respectively refer to the number of failed executions and passed executions that cover cv_i. They are calculated using Eqs. (3) and (4).

$$a_{ef}(t_j, cv_i) = |\{t_j \mid cv_i \in \mathit{failed}(t_j)\}| \tag{3}$$
$$a_{ep}(t_j, cv_i) = |\{t_j \mid cv_i \in \mathit{passed}(t_j)\}|. \tag{4}$$

According to Eq. (2), for the coverage vectors in the motivating example in Sec. 2, the failing extent is calculated as $F_{\mathrm{SBI}}(cv_1) = 1.00$, $F_{\mathrm{SBI}}(cv_2) = 0.50$, $F_{\mathrm{SBI}}(cv_3) = 0.25$, $F_{\mathrm{SBI}}(cv_4) = 0.00$, and $F_{\mathrm{SBI}}(cv_5) = 0.00$, respectively. Since cv_1, cv_2, and cv_3 have ever been exercised in a failed execution, thus these three coverage vectors are failed coverage vectors. On the contrary, cv_4 and cv_5 are passed coverage vectors.

3.5. *Calculation of suspiciousness scores*

Following the popular fault localization manner, in our framework, we use the suspiciousness score of a basic block to estimate the extent of a basic block being related to faults. In this paper, we denote the suspiciousness score of the block b_i as $S(b_i)$.

When employing a base technique to synthesize a fault localization technique in our framework, we adapt $S_T(b_i)$ to calculate the suspiciousness score. The suspiciousness score of a basic block b_i is calculated as Eq. (5).

$$S_T(b_i) = f_T(a_{np}(cv_j, b_i), a_{nf}(cv_j, b_i), a_{ep}(cv_j, b_i), a_{ef}(cv_j, b_i)). \tag{5}$$

In Eq. (5), f_T is the formula of the base technique T, $a_{np}(cv_j, b_i)$, $a_{nf}(cv_j, b_i)$, $a_{ep}(cv_j, b_i)$, and $a_{ef}(cv_j, b_i)$ mean the number of passed coverage vectors that do not cover b_i, the number of failed coverage vectors that do not cover b_i, the number of passed coverage vectors that cover b_i, and the number of failed coverage vectors that cover b_i, respectively.

For example, if we use SBI as the base technique, we can adopt the formula of SBI to calculate the suspiciousness score. It is shown as Eq. (6).

$$S_{\mathrm{SBI}}(b_i) = \frac{a_{ef}(cv_j, b_i)}{a_{ef}(cv_j, b_i) + a_{ep}(cv_j, b_i)}. \tag{6}$$

Here, $S(b_i)$ represents the extent of how much a basic block is related to faults. The greater the value, the more the basic block will be related to faults. Let us recall the motivating example in Sec. 2 and revisit the suspiciousness scores calculated in Sec. 2. According to Eq. (6), the suspiciousness scores for b_1 and b_{10} are calculated as $\frac{3}{3+2} = 0.60$. The suspiciousness score for b_2 is calculated as $\frac{2}{2+1} = 0.67$. The suspiciousness score for b_3 is calculated as $\frac{1}{1+0} = 1.00$. The suspiciousness scores for b_4, b_6 and b_8 are calculated as $\frac{1}{1+1} = 0.50$. The suspiciousness scores for b_5 and b_9 are calculated as $\frac{0}{0+1} = 0.00$.

Till now, we have assigned a suspiciousness score to each basic block. We then sort all the basic blocks in descending order of their suspiciousness scores, to form a ranked list of all basic blocks. All the basic blocks not exercised in any execution will be grouped to form a new basic block, which is added to the end of the ranked list.

After all the blocks are sorted according to their suspiciousness of being related to faults and form a list, programmers may search along the generated list for the fault. Particularly, when some basic blocks have identical suspiciousness scores, we use Eq. (7) to break tie.

$$C(b_i) = \frac{\sum_{cv_j(b_i)=1} [F_T(cv_j)]}{|\{cv_j \,|\, F_T(cv_j) > 0 \wedge cv_j(b_i) = 1\}|}. \tag{7}$$

Equation (7) calculates the average failing extent of the coverage vectors that exercising basic block b_i. The rationale is that for two basic blocks having identical probability of causing failure, we deem the one whose appearance in a path having higher chance to reveal failures as more fault-relevant.

Table 1. The 34 statement-level fault-localization techniques.

Name	Failing extent of coverage vectors	Suspiciousness scores of blocks
Jaccard [33]	$t_{ef}/(t_{nf}+t_{ef}+t_{ep})$	$c_{ef}/(c_{nf}+c_{ef}+c_{ep})$
Anderberg [34]	$t_{ef}/(t_{ef}+2(t_{nf}+t_{ep}))$	$c_{ef}/(c_{ef}+2(c_{nf}+c_{ep}))$
Sørensen-Dice [35]	$t_{ef}/(t_{ef}+t_{nf}+t_{ep})$	$c_{ef}/(c_{ef}+c_{nf}+c_{ep})$
Dice [36]	$2t_{ef}/(t_{ef}+t_{nf}+t_{ep})$	$2c_{ef}/(c_{ef}+c_{nf}+c_{ep})$
Kulczynskil [37]	$t_{ef}/(t_{nf}+t_{ep})$	$c_{ef}/(c_{nf}+c_{ep})$
Kulczynski2 [37]	$\frac{1}{2}\left(\frac{t_{ef}}{t_{ef}+t_{nf}}+\frac{t_{ef}}{t_{ef}+t_{ep}}\right)$	$\frac{1}{2}\left(\frac{c_{ef}}{c_{ef}+c_{nf}}+\frac{c_{ef}}{c_{ef}+c_{ep}}\right)$
Russell and Rao [38]	$t_{ef}/(t_{ef}+t_{nf}+t_{ep}+t_{np})$	$c_{ef}/(c_{ef}+c_{nf}+c_{ep}+c_{np})$
Hamann [37]	$(t_{ef}+t_{np}-t_{nf}-t_{ep})/(t_{ef}+t_{nf}+t_{ep}+t_{np})$	$(c_{ef}+c_{np}-c_{nf}-c_{ep})/(c_{ef}+c_{nf}+c_{ep}+c_{np})$
Simple Matching [39]	$(t_{ef}+t_{np})/(t_{ef}+t_{nf}+t_{ep}+t_{np})$	$(c_{ef}+c_{np})/(c_{ef}+c_{nf}+c_{ep}+c_{np})$
Sokal [37]	$2(t_{ef}+t_{np})/(2t_{ef}+2t_{np}+t_{nf}+t_{ep})$	$2(c_{ef}+c_{np})/(2c_{ef}+2c_{np}+c_{nf}+c_{ep})$
M1 [40]	$(t_{ef}+t_{np})/(t_{nf}+t_{ep})$	$(c_{ef}+c_{np})/(c_{nf}+c_{ep})$
M2 [40]	$t_{ef}/(t_{ef}+t_{np}+2t_{nf}+2t_{ep})$	$c_{ef}/(c_{ef}+c_{np}+2c_{nf}+2c_{ep})$
Rogers and Tanimoto [41]	$(t_{ef}+t_{np})/(t_{ef}+t_{nf}+2t_{nf}+2t_{ep})$	$(c_{ef}+c_{np})/(c_{ef}+c_{nf}+2c_{nf}+2c_{ep})$
Goodman [42]	$(2t_{ef}-t_{nf}-t_{ep})/(2t_{ef}+t_{nf}+t_{ep})$	$(2c_{ef}-c_{nf}-c_{ep})/(2c_{ef}+c_{nf}+c_{ep})$
Hamming [43]	$t_{ef}+t_{np}$	$c_{ef}+c_{np}$
Euclid [44]	$\sqrt{t_{ef}+t_{np}}$	$\sqrt{c_{ef}+c_{np}}$
Ochiai [45]	$\frac{t_{ef}}{\sqrt{(t_{ef}+t_{nf})(t_{ef}+t_{ep})}}$	$\frac{c_{ef}}{\sqrt{(c_{ef}+c_{nf})(c_{ef}+t_{ep})}}$
Overlap [44]	$t_{ef}/min(t_{ef},t_{nf},t_{ep})$	$c_{ef}/min(c_{ef},t_{nf},c_{ep})$
Tarantula [15]	$\frac{t_{ef}/(t_{ef}+t_{nf})}{t_{ef}/(t_{ef}+t_{nf})+t_{ep}/(t_{ep}+t_{np})}$	$\frac{c_{ef}/(c_{ef}+c_{nf})}{c_{ef}/(c_{ef}+c_{nf})+c_{ep}/(c_{ep}+c_{np})}$
Zoltar [46]	$\frac{t_{ef}}{t_{ef}+t_{nf}+t_{ep}+\frac{10000t_{nf}t_{ep}}{t_{ef}}}$	$\frac{c_{ef}}{c_{ef}+c_{nf}+c_{ep}+\frac{10000c_{nf}c_{ep}}{c_{ef}}}$
Ample [47]	$\left\lvert\frac{t_{ef}}{t_{ef}+t_{nf}}-\frac{t_{ep}}{t_{ep}+t_{np}}\right\rvert$	$\left\lvert\frac{c_{ef}}{c_{ef}+c_{nf}}-\frac{c_{ep}}{c_{ep}+t_{np}}\right\rvert$
Wong1 [48]	t_{ef}	c_{ef}
Wong2 [48]	$t_{ef}-t_{ep}$	$c_{ef}-c_{ep}$
Wong3 [48]	$t_{ef}-\begin{cases}t_{ep} & t_{ep}\leq 2\\ 2+0.1(t_{ep}-2) & 2\leq t_{ep}\leq 10\\ 2.8+0.001(t_{ep}-10) & t_{ep}\geq 10\end{cases}$	$c_{ef}-\begin{cases}c_{ep} & c_{ep}\leq 2\\ 2+0.1(c_{ep}-2) & 2\leq c_{ep}\leq 10\\ 2.8+0.001(c_{ep}-10) & c_{ep}\geq 10\end{cases}$
Ochiai2 [39]	$\frac{t_{ef}t_{np}}{\sqrt{(t_{ef}+t_{ep})(t_{np}+t_{nf})(t_{ef}+t_{nf})(t_{ep}+t_{np})}}$	$\frac{c_{ef}c_{np}}{\sqrt{(c_{ef}+c_{ep})(c_{np}+c_{nf})(c_{ef}+c_{nf})(c_{ep}+c_{np})}}$
Geometric Mean [49]	$\frac{t_{ef}t_{np}-t_{nf}t_{ep}}{\sqrt{(t_{np}+t_{ep})(t_{np}+t_{nf})(t_{ef}+t_{nf})(t_{ep}+t_{np})}}$	$\frac{c_{ef}c_{np}-c_{nf}c_{ep}}{\sqrt{(c_{ef}+c_{ep})(c_{np}+c_{nf})(c_{ef}+c_{nf})(c_{ep}+c_{np})}}$
Harmonic Mean [50]	$\frac{(t_{ef}t_{np}-t_{nf}t_{ep})((t_{ef}+t_{ep})(t_{np}+t_{nf})+(t_{ef}+t_{nf})(t_{ep}+t_{np}))}{(t_{ef}+t_{ep}(c_i))(t_{np}+t_{nf})(t_{ef}+t_{nf})(t_{ep}+t_{np})}$	$\frac{(c_{ef}c_{np}-c_{nf}c_{ep})((c_{ef}+c_{ep})(c_{np}+c_{nf})+(c_{ef}+c_{nf})(c_{ep}+c_{np}))}{(c_{ef}+c_{ep}(b_j))(c_{np}+c_{nf})(c_{ef}+c_{nf})(c_{ep}+c_{np})}$
Arithmetic Mean [50]	$\frac{2t_{ef}t_{np}-2t_{nf}t_{ep}}{(t_{ef}+t_{ep})(t_{np}+t_{nf})+(t_{ef}+t_{nf})(t_{ep}+t_{np})}$	$\frac{2c_{ef}c_{np}-2c_{nf}c_{ep}}{(c_{ef}+c_{ep})(c_{np}+c_{nf})+(c_{ef}+c_{nf})(c_{ep}+c_{np})}$
Cohen [51]	$\frac{2t_{ef}t_{np}-2t_{nf}t_{ep}}{(t_{ef}+t_{ep})(t_{np}+t_{nf})+(t_{ef}+t_{nf})(t_{ep}+t_{np})}$	$\frac{2c_{ef}c_{np}-2c_{nf}c_{ep}}{(c_{ef}+c_{ep})(c_{np}+c_{nf})+(c_{ef}+c_{nf})(c_{ep}+c_{np})}$
Scott [52]	$\frac{4t_{ef}t_{np}-4t_{nf}t_{ep}-(t_{nf}-t_{ep})^2}{(2t_{ef}+t_{nf}+t_{ep})(2t_{np}+t_{nf}+t_{ep})}$	$\frac{4c_{ef}c_{np}-4c_{nf}c_{ep}-(c_{nf}-c_{ep})^2}{(2c_{ef}+c_{nf}+c_{ep})(2c_{np}+c_{nf}+c_{ep})}$
Fleiss [53]	$\frac{4t_{ef}t_{np}-4t_{nf}t_{ep}-(t_{nf}-t_{ep})^2}{(2t_{ef}+t_{nf}+t_{ep})+(2t_{np}+t_{nf}+t_{ep})}$	$\frac{4c_{ef}c_{np}-4c_{nf}c_{ep}-(c_{nf}-c_{ep})^2}{(2c_{ef}+c_{nf}+c_{ep})+(2c_{np}+c_{nf}+c_{ep})}$
Rogot1 [50]	$\frac{1}{2}\left(\frac{t_{ef}}{2t_{ef}+t_{nf}+t_{ep}}+\frac{t_{np}}{2t_{np}+t_{nf}+t_{ep}}\right)$	$\frac{1}{2}\left(\frac{c_{ef}}{2c_{ef}+c_{nf}+c_{ep}}+\frac{c_{np}}{2c_{np}+c_{nf}+c_{ep}}\right)$
Rogot2 [50]	$\frac{1}{4}\left(\frac{t_{ef}}{t_{ef}+t_{ep}}+\frac{t_{ef}}{t_{ef}+t_{nf}}+\frac{t_{np}}{t_{np}+t_{ep}}+\frac{t_{np}}{t_{np}+t_{nf}}\right)$	$\frac{1}{4}\left(\frac{c_{ef}}{c_{ef}+c_{ep}}+\frac{c_{ef}}{c_{ef}+c_{nf}}+\frac{c_{np}}{c_{np}+c_{ep}}+\frac{c_{np}}{c_{np}+c_{nf}}\right)$
SBI [21]	$t_{ef}/(t_{ef}+t_{ep})$	$c_{ef}/(c_{ef}+c_{ep})$

For example, in Fig. 1, basic blocks b_1 and b_{10} form a tie. With the base techniques of SBI, we calculate that $C(b_1) = C(b_{10}) = \frac{1+0.5+0.25}{3} = 0.58$. As a result, the tie still cannot be break, and thus b_1 and b_{10} are evaluated as a whole. Finally, we need to examine 40% of all code to locate the fault.

3.6. *More illustrations*

In Table 1, we list out how to apply the 34 existing techniques (SBI and the 33 techniques listed in [18, 19]) in out framework (on calculation of failing extent and suspiciousness scores).

In Fig. 3, due to the literal limitation, we use t_{np}, t_{nf}, t_{ep}, and t_{ef} to stand for the four parameters $a_{np}(t_j, cv_i)$, $a_{nf}(t_j, cv_i)$, $a_{ep}(t_j, cv_i)$, and $a_{ef}(t_j, cv_i)$, respectively. We use c_{np}, c_{nf}, c_{ep}, and c_{ef} to stand for the four parameters $a_{np}(cv_j, b_i)$, $a_{nf}(cv_j, b_i)$, $a_{ep}(cv_j, b_i)$, and $a_{ef}(cv_j, b_i)$, respectively.

4. Empirical Evaluation

In this section, we conduct a controlled experiment to evaluate the effectiveness of our framework, and compare the synthesized techniques with base techniques and peer techniques on fault localization effectiveness, over different programs.

4.1. *Subject programs*

In this paper, we use the seven Siemens programs and three median-sized real-life UNIX utility programs to evaluate our framework. Each of the ten programs is attached with several faulty versions (each contains one fault) and a test pool (both downloaded from the Software-artifact Infrastructure Repository, also SIR for abbreviation [54]). All these subjects have been used in previous studies [4, 6, 55].

Table 2 shows the statistics of the subject programs used in the experiments. Take the program *flex* for example, there are 20 different faulty versions, and 567 test cases attached with it.

Table 2. Statistics of subject programs.

Type	Programs	Faulty versions	Test cases	Description
Siemens	print_tokens	7	4130	lexical analyzer
	print_tokens2	10	4115	lexical analyzer
	replace	29	5542	pattern replacement
	schedule	9	2650	priority scheduler
	schedule2	9	2650	priority scheduler
	tcas	40	1578	altitude separation
	tot_info	23	1054	information measure
UNIX	flex	56	567	lexical parser
	grep	21	809	text processor
	gzip	18	213	compressor
	in total	222		

Table 3. The mean code examining effort to locate a fault for each of the 16 approaches.

	Code examining effort										
Approaches	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
F_{Ochiai}-S_{Ochiai}	0.0%	43.7%	53.7%	61.1%	66.1%	70.4%	71.7%	73.1%	75.7%	77.1%	100%
F_{Jaccard}-S_{Ochiai}	0.0%	43.7%	53.7%	61.1%	66.1%	70.4%	71.7%	73.1%	75.7%	77.1%	100%
$F_{\text{Tarantula}}$-S_{Ochiai}	0.0%	43.7%	53.7%	61.1%	65.3%	69.9%	71.7%	73.1%	75.4%	77.1%	100%
F_{SBI}-S_{Ochiai}	0.0%	43.7%	53.7%	61.1%	65.3%	69.9%	71.7%	73.1%	75.4%	77.1%	100%
F_{Ochiai}-S_{Jaccard}	0.0%	41.2%	50.3%	58.9%	63.8%	67.1%	69.5%	72.1%	75.7%	75.9%	100%
F_{Jaccard}-S_{Jaccard}	0.0%	41.2%	50.3%	58.9%	63.8%	67.1%	69.5%	72.1%	75.7%	75.9%	100%
$F_{\text{Tarantula}}$-S_{Jaccard}	0.0%	41.2%	50.3%	58.9%	63.8%	66.5%	69.5%	72.1%	75.7%	75.9%	100%
F_{SBI}-S_{Jaccard}	0.0%	41.2%	50.3%	58.9%	63.8%	66.5%	69.5%	72.1%	75.7%	75.9%	100%
F_{Ochiai}-$S_{\text{Tarantula}}$	0.0%	30.9%	45.0%	54.5%	59.1%	65.2%	69.3%	72.3%	72.7%	72.9%	100%
F_{Jaccard}-$S_{\text{Tarantula}}$	0.0%	31.1%	44.8%	54.5%	59.1%	65.2%	69.3%	72.3%	72.7%	72.7%	100%
$F_{\text{Tarantula}}$-$S_{\text{Tarantula}}$	0.0%	30.0%	44.3%	54.0%	58.7%	65.2%	69.3%	72.3%	72.7%	72.7%	100%
F_{SBI}-$S_{\text{Tarantula}}$	0.0%	30.0%	44.3%	54.0%	58.7%	65.2%	69.3%	72.3%	72.7%	72.7%	100%
F_{Ochiai}-S_{SBI}	0.0%	30.8%	44.8%	54.4%	59.1%	65.2%	69.3%	72.3%	72.7%	72.9%	100%
F_{Jaccard}-S_{SBI}	0.0%	30.9%	44.6%	54.3%	58.9%	65.0%	69.3%	72.1%	72.7%	72.9%	100%
$F_{\text{Tarantula}}$-S_{SBI}	0.0%	29.8%	44.2%	53.8%	58.5%	64.8%	68.9%	72.1%	72.4%	72.4%	100%
F_{SBI}-S_{SBI}	0.0%	29.8%	44.2%	53.8%	58.5%	65.0%	69.2%	72.1%	72.7%	72.7%	100%

4.2. *Subject techniques*

In Table 3, all the 34 existing techniques could be used as base techniques. In our experiment, we only select four techniques Ochiai [22], Jaccard [22], Tarantula [15], and SBI [21] as base techniques to validate our framework, because these four techniques are more representative and widely used as peer techniques in previous studies. For example, Ochiai and Jaccard are evaluated to be very effective in previous studies [22], Tarantula is one of the pioneer and well-known technique and has a lot of variants [8, 21], the original technique of SBI [21] (CBI [3]) is a classical predicate-level technique and we are interested in comparing our framework with those of predicate-based techniques.

Hao *et al.* [6] propose a similarity-aware fault localization technique SAFL, which takes test cases as fuzzy set to deal with the similarities among test cases, and calculates suspiciousness scores based on the probability theory. Masri and Assi [31] propose a technique, denoted as ICST10, to cleans test suites from coincidental correctness and further enhance the fault localization. These approaches are close, in terms of basic idea, to our approach. Since they cannot be included into the $S_T(a_{np}, a_{nf}, a_{ep}, a_{ef})$ settings, we compare the best technique synthesized in our framework with them, on fault localization effectiveness.

4.3. *Evaluation metrics*

In previous studies, the evaluation metrics is often defined as the ratio of the statements (program lines) examined before reaching a fault [48, 4], when searching along the list of program entities with the decreasing order of suspiciousness scores.

Execution spectra is represented as the coverage of execution traces, and the coverage information indicated which statements are executed. Since statements in a basic block will always (in some rare cases not) share identical execution spectra, they are indistinguishable one another, at the view point of a CBFL technique. For these reasons, statements may not be always a proper unit for fault localization. In our study, we use basic block as the natural program unit, calculate suspiciousness scores for them, generate a ranked list for them, and search along the ranked list of them for *faulty basic blocks* (which containing the faulty statements). In our study, the effectiveness metrics is defined as the ratio of the basic block checked before reaching a faulty one. It is also referred to as the *code examining effort* in the rest of the paper.

4.4. *Experiment setup*

We set up our experiments on the platform of *ubuntu* 10.4. The compiler is *gcc*-4.4.1 and the component *gcov* is used to collect the execution spectra. In our experiment, we follow previous studies [48, 8, 4] to exclude those faulty versions whose fault cannot be manifested with all test cases in the test suite. Further, we exclude those program annotations, blank lines, function declarations, variable declarations and so on statements because they are not executable and cannot be processed by CBFL techniques [4]. Besides, for faults on a non-executable statement or a statement of which the execution spectra cannot be collected, such as a macro definition or a statement-omission fault, we take after previous work [19] to mark the directly affected or closest adjacent executable statement as the fault and evaluate the effectiveness of locating its belonging basic block.

In our framework, the equation to calculate the failing extent of a coverage vector and the equation to calculate the suspiciousness score of a block are determined according to the base techniques specified. In our controlled experiments, we select the formulas which are used in Ochiai, Jaccard, Tarantula, and SBI to calculate the failing extent and suspiciousness score. We use F_T and S_T to denote the formulas of using technique T ($T \in \{\text{Ochiai}, \text{Jaccard}, \text{Tarantula}, \text{SBI}\}$) to calculate failing extent and suspiciousness scores, respectively. Section 4.5 shows the result of these four approaches.

Since the failing extent and suspiciousness scores are calculated separately, we do not limit to apply same formula for them. For example, we are also interested to see the result of employing SBI to calculate failing extent and employing Jaccard to calculate the suspiciousness score in our framework. Thus we can generate 16 different combinations for F_T and $S_{T'}$ where $T, T' \in \{\text{Ochiai}, \text{Jaccard}, \text{Tarantula}, \text{SBI}\}$. The corresponding results are shown in Sec. 4.6.

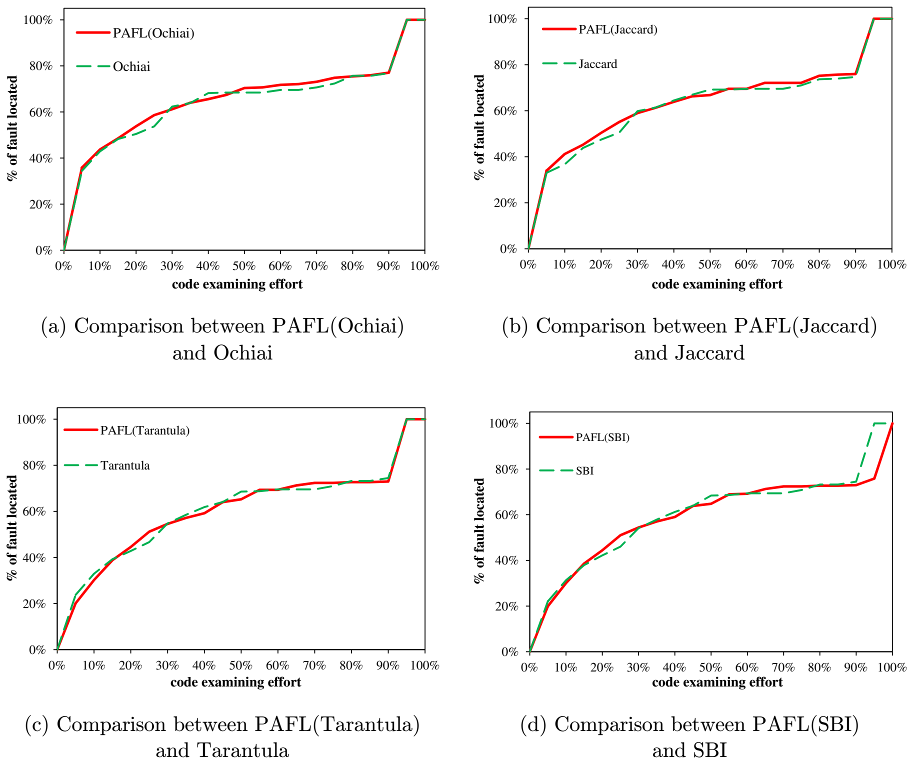
From the previous tests, we can decide the best technique synthesized in our framework. Recall that we also consider two close related work SAFL and ICST10, yet they cannot be used as base techniques of our framework and thus cannot be evaluated in the previous tests. We further compare the best technique synthesized in our framework with SAFL and ICST10, on the fault localization effectiveness, in Sec. 4.7.

4.5. *Results on four base techniques*

In this section, we compare the techniques synthesized in our framework with the base techniques. We select four base techniques, Ochiai, Jaccard, Tarantula, and SBI, in this test. For example, when using Ochiai as the base technique, the formula for calculating failing extent and suspiciousness scores are denoted as F_{Ochiai} and S_{Ochiai}. We use F_{Ochiai}-S_{Ochiai} to denote such an approach.

We apply four base techniques to synthesize four approaches, F_{Ochiai}-S_{Ochiai}, F_{Jaccard}-S_{Jaccard}, $F_{\text{Tarantula}}$-$S_{\text{Tarantula}}$, and F_{SBI}-S_{SBI}, in our framework. We respectively compare their fault localization results with the results of Ochiai, Jaccard, Tarantula, and SBI. To ease the explanation, we also denote F_{Ochiai}-S_{Ochiai} as PAFL (Ochiai), F_{Jaccard}-S_{Jaccard} as PAFL(Jaccard), $F_{\text{Tarantula}}$-$S_{\text{Tarantula}}$ as PAFL(Tarantula), and F_{SBI}-S_{SBI} as PAFL(SBI). The experiment results are shown in Fig. 3.

In Fig. 3, there are four plots, which represent the comparisons between PAFL(Ochiai) and Ochiai, PAFL(Jaccard) and Jaccard, PAFL(Tarantula) and



(a) Comparison between PAFL(Ochiai) and Ochiai

(b) Comparison between PAFL(Jaccard) and Jaccard

(c) Comparison between PAFL(Tarantula) and Tarantula

(d) Comparison between PAFL(SBI) and SBI

Fig. 3. Comparison between techniques synthesized into PAFL with base techniques.

Tarantula, PAFL(SBI) and SBI. Take the first plot to illustrate, the X-coordinate shows the code examining effort, the Y-coordinate shows the faults located within the given (by X) code examining effort. For example, in the first plot, the point $(10\%, 43.7\%)$ is on the curve of PAFL(Ochiai). It shows that when examining 10% basic blocks (in the suggested ranked list order generated by the PAFL(Ochiai) approach) in each of the faulty version, faulty blocks in 43.7% of the 222 faulty versions in total can be reached. For the curve of Ochiai, the corresponding point is $(10\%, 42.9\%)$. It means that within the same code examining effort, Ochiai can locate 42.9% of the 222 faults.

We first focus on the two state-of-the-art base techniques Ochiai and Jaccard. From Plot (a), we have the observation that the curve of PAFL(Ochiai) is always above or overlap the curve of Ochiai (except the region of $(40\%, 60\%)$). It means that the fault localization effectiveness of PAFL(Ochiai) is always higher than that of Ochiai (except a small code examining region). From Plot (b), we have similar observation, and know that the fault localization effectiveness of PAFL(Jaccard) is always higher than that of Jaccard (except a small code examining region). When we move to the other two base techniques Tarantula and SBI, we observe from Plot(c) and Plot(d) that PAFL(Tarantula) and PAFL(SBI) have a comparable effectiveness with Tarantula and SBI, respectively.

Our preliminary observation is that our framework is very effective in synthesizing a more effective fault localization technique on Ochiai and Jaccard, while cannot improve much on Tarantula and SBI. We summarize our observation in this test as "our framework is more effective on more effectiveness fault localization techniques".

4.6. *Results on mixed usage of four base techniques*

In the previous section, we have investigated the results of F_{Ochiai}-S_{Ochiai}, F_{Jaccard}-S_{Jaccard}, $F_{\text{Tarantula}}$-$S_{\text{Tarantula}}$, and F_{SBI}-S_{SBI} in our framework. In each of these four approaches, the calculation of failing extent and suspiciousness scores use the core formula of the base technique adopted. Further, we are also interested in the mixed usage of the four base techniques when calculating failing extent and suspiciousness scores. For example, what will the result of F_{Jaccard}-S_{Ochiai} looks like, when comparing with F_{Jaccard}-S_{Jaccard} and F_{Ochiai}-S_{Ochiai}? In this section, we will show the results of employing different base techniques to calculate the failing extent and suspiciousness scores.

The calculation of failing extent and suspiciousness scores can be conducted independently. Since we have four different base techniques to employ in the calculation of failing extent and suspiciousness scores, there will be 16 different combinations. For different approaches, Fig. 4 shows the percentage of faults located within different code examining effort, averaging results over different programs. To do that, we use the effectiveness metrics to evaluate the fault localization effectiveness of a synthesized technique in locating faults in every faulty versions, calculate

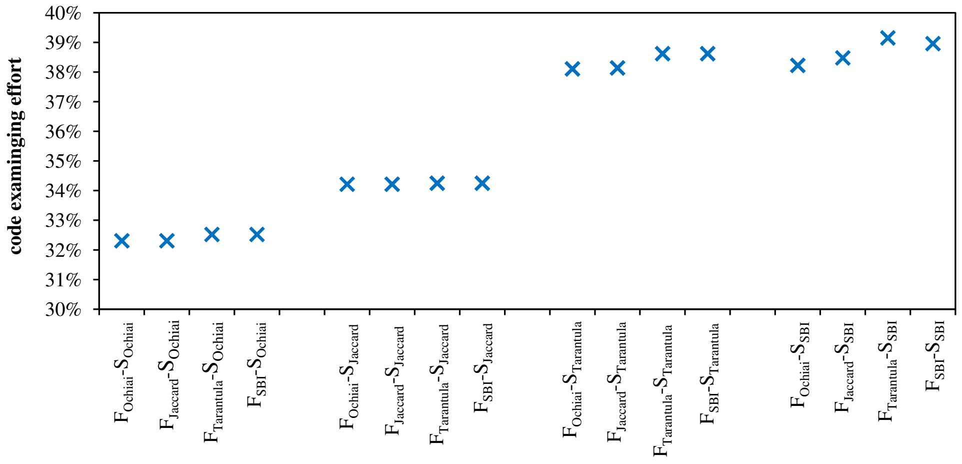


Fig. 4. The mean code examining effort to locate a fault for each of the 16 approaches.

the percentage of faults located within specific code examining effort for every programs, and generate the in average percentage of faults located with specific code examining effort. We do that because such averaging may alleviate the bias from specific programs.

The 16 approaches in Fig. 4 are denoted in a form of F_T-$S_{T'}$. F_T and $S_{T'}$ refer to the formulas adopted to calculate the failing extent and the suspiciousness scores, respectively. Take the $F_{\mathrm{SBI}} - S_{\mathrm{Tarantula}}$ for illustration. It means that we adopt the formula of SBI to calculate the failing extent, and the formula of Tarantula to calculate the suspiciousness scores. Other approaches can be explained similarly.

From Fig. 4, we can observe that applying different base techniques to calculate failing extent or suspiciousness scores can result in different fault localization effectiveness. For example, when examining up to 10% basic blocks, F_{Jaccard}-S_{Ochiai}, $F_{\mathrm{Tarantula}}$-S_{Ochiai}, F_{Jaccard}-S_{SBI}, and $F_{\mathrm{Tarantula}}$-S_{SBI} can locate faults in 43.7%, 43.7%, 30.9%, and 29.8% faulty versions, respectively. Further, when using a specific technique to calculate the suspiciousness score, applying different base techniques to calculate the failing extent can result in different fault localization effectiveness. For example, when using Ochiai to calculate the suspiciousness score, applying Ochiai or Jaccard to calculate the failing extent is more promising than applying Tarantula or SBI. On the other hand, when using a specific technique to calculate the failing extent, applying different base techniques to calculate the suspiciousness score can also result in different fault localization effectiveness. For example, when using Ochiai to calculate the failing extent, applying Ochiai or Jaccard to calculate the suspiciousness score is more promising than applying Tarantula or SBI.

Another observation is that the calculation of suspiciousness scores has more impact on the fault localization effectiveness than the calculation of failing extent does. For example, when applying Ochiai to calculate the suspiciousness score, whatever a technique is chosen to calculate the failing extent, the corresponding approach can locate faults in at least 43.7% faulty versions, within the 10% code examining effort. With the same code examining effort, when applying SBI to calculate the suspiciousness score, whatever a technique is chosen to calculate the failing extent, the corresponding approach can locate faults in at most 30.9% faulty versions. Further, a trend is that generally, applying Ochiai to calculate suspiciousness scores can be more promising than applying Jaccard, and in turn more promising than applying Tarantula and SBI, whatever a technique is used to calculate the failing extent. Another trend is that when applying a specific technique to calculate suspiciousness scores, applying Ochiai to calculate failing extent is mostly more promising than applying Jaccard, and in turn more promising than applying Tarantula and SBI.

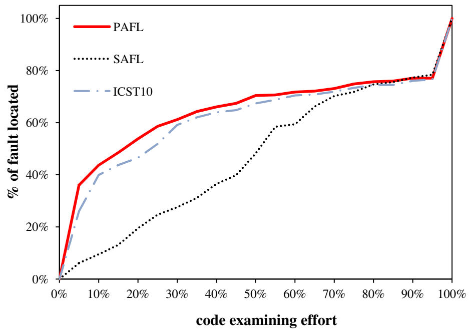
To confirm our observation on the dominant effect of using different techniques to calculate suspiciousness score, we further use Fig. 4 to investigate. Figure 4 shows the *mean* code examining effort to locate a fault for each of the 16 approaches. From this figure, the dominant effect of using different techniques to calculate suspiciousness scores can be clearly observed.

As a summary, we claim that both the calculation of failing extent and suspiciousness scores have impact on the fault localization effectiveness of the techniques synthesized using our framework, while the latter has a dominant effect. Further, either on calculation of failing extent or on calculation of suspiciousness scores, applying Ochiai is more promising than applying Jaccard, and in turn more promising than applying Tarantula and SBI. As a result, F_{Ochiai}-S_{Ochiai}, PAFL(Ochiai) for short, is the best technique synthesized in our framework. In the next section, we will compare PAFL(Ochiai) with other peer techniques.

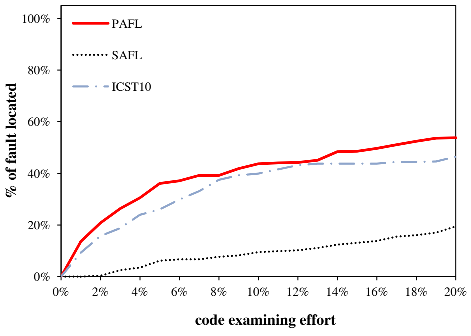
4.7. *Results comparison of PAFL(Ochiai), SAFL, and ICST*10

In this section, we compare the fault localization effectiveness of PAFL(Ochiai), PAFL in short in particular for this section, which means applying Ochiai as the base technique to synthesize a new fault localization technique in our framework, with SAFL [6] and ICST10 [31].

We first compare the overall effectiveness of the three techniques in Fig. 5. In the plots of Fig. 5, the X-coordinate means the code examining effort in each faulty version, the Y-coordinate shows the percentage of faulty versions, faults in which can be located within the code examining effort specified by the X-coordinate. Note that in the plots, all the curves start from the (0%, 0%) point and end at the (100%, 100%) point. It means that when examining no code, none of the faults can be located; while all faults can be located when examining all the code. From Plot (a) of Fig. 5, we observe that at every checkpoints, PAFL is always more effective than the other techniques. For example, on average, by examining up to 5% of all the code in each faulty version, PAFL can locate faults in 36.1% of all faulty versions, SAFL can locate 6.1%, and ICST10 can locate 26.0% accordingly. By examining up to 10% of



(a)



(b)

Fig. 5. Overall results in (a) full range [0%, 100%] and (b) zoom-in range of [0%, 20%].

all the code, PAFL can locate faults in 43.7% of all faulty versions, SAFL can locate 9.5%, and ICST10 can locate 39.9% accordingly. It shows that PAFL has an overall better effectiveness than the other techniques studied.

At the same time, since previous study has pointed out that the top 20% code examining range is more important than the others [56], we also zoom in the results to the [0%, 20%] range and show it in Plot (b) of Fig. 5. Plot (b) shows a similar phenomenon as Plot (a). In the whole range of [0%, 20%], the effectiveness of PAFL is always better than those of SAFL and ICST10. For example, on average, by examining up to 1% of the code in all the faulty versions, PAFL can locate faults in 13.6% of all faulty versions, while SAFL can locate 0%, and ICST10 can locate 9.3% accordingly. By examining up to 3% of the code, PAFL can locate faults in 26.6% of all faulty versions, while SAFL can locate 2.5%, and ICST10 can locate 18.8% accordingly.

To sufficiently compare different techniques, we further use a popular statistics metrics, box-whisker plot, to percent the results. The box-whisker plot of the effectiveness of each technique is shown in Fig. 6. In Fig. 6, each box-whisker stands for

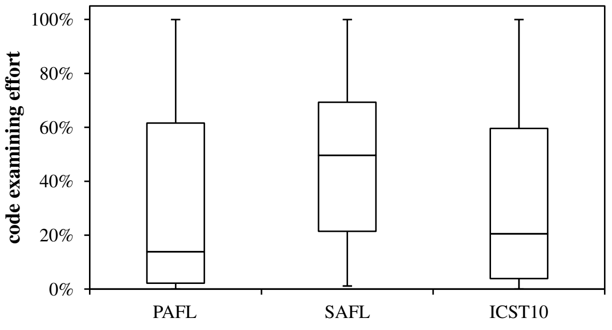


Fig. 6. The box-whisker plot for overall effectiveness.

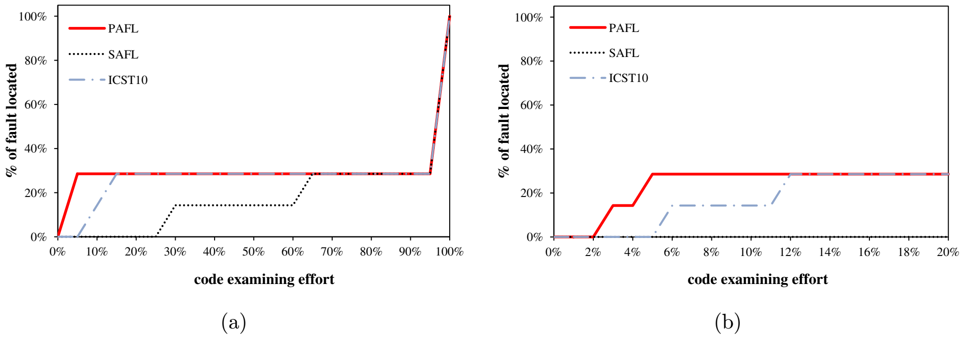


Fig. 7. Results on *print_tokens* in (a) full range [0%, 100%] and (b) zoom-in range of [0%, 20%].

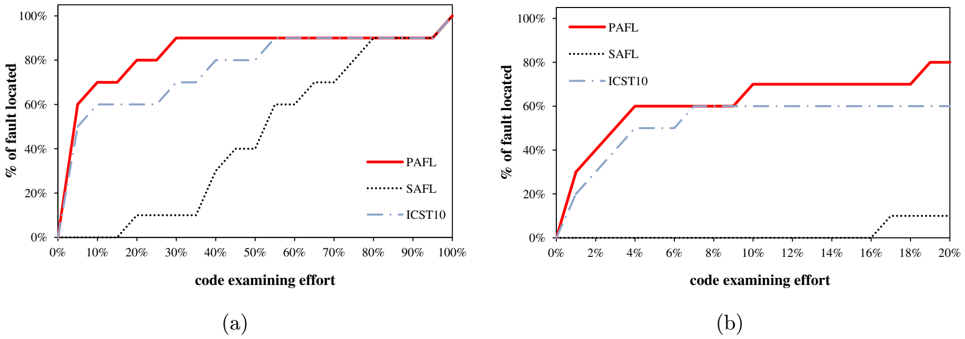


Fig. 8. Results on *print_tokens*2 in (a) full range [0%, 100%] and (b) zoom-in range of [0%, 20%].

effectiveness statistics of a technique (PAFL, SAFL, and ICST10). The height of the box spans the central 50% of the data and its upper bound and lower bound mark the upper (75% percentile) and lower (25% percentile) quartiles, respectively. The middle line in the box represents the median value of fault localization effectiveness.

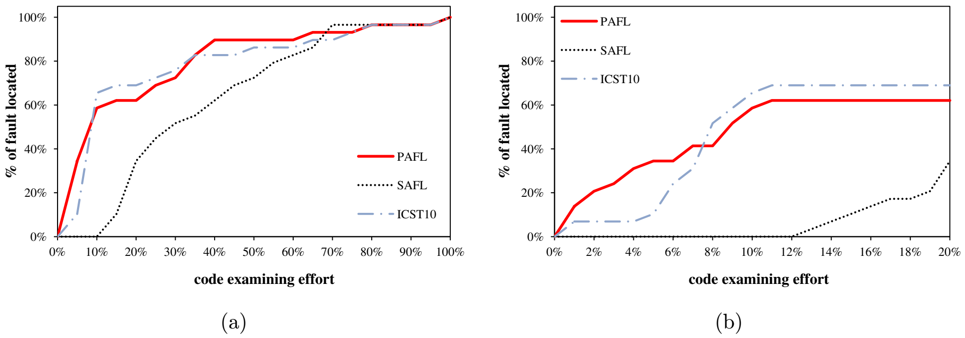


Fig. 9. Results on *replace* in (a) full range [0%, 100%] and (b) zoom-in range of [0%, 20%].

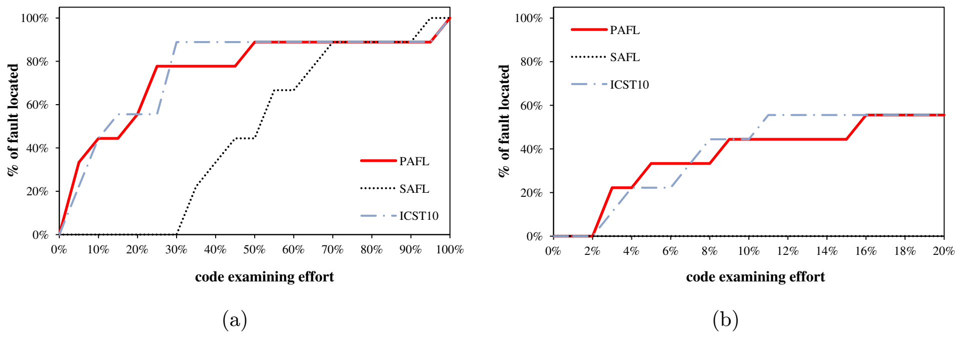


Fig. 10. Results on *schedule* in (a) full range [0%, 100%] and (b) zoom-in range of [0%, 20%].

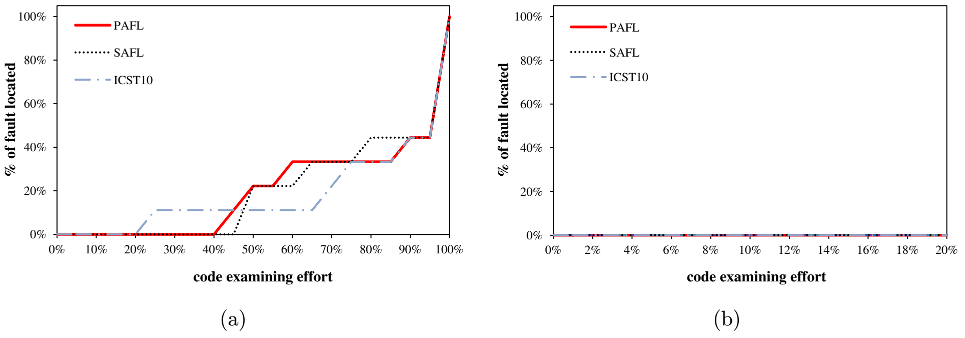


Fig. 11. Results on *schedule*2 in (a) full range [0%, 100%] and (b) zoom-in range of [0%, 20%].

The top of the upper whisker and the bottom of the lower whisker indicate the worst case and best case of locating faults, respectively. From Fig. 6, we have the observation that PAFL performs better than SAFL and ICST10 at most aspects. For example, when we checking the mean case, PAFL takes in average 13.9% code examining effort to locate a fault, while SAFL and ICST10 needs 45.6%, and 20.5%,

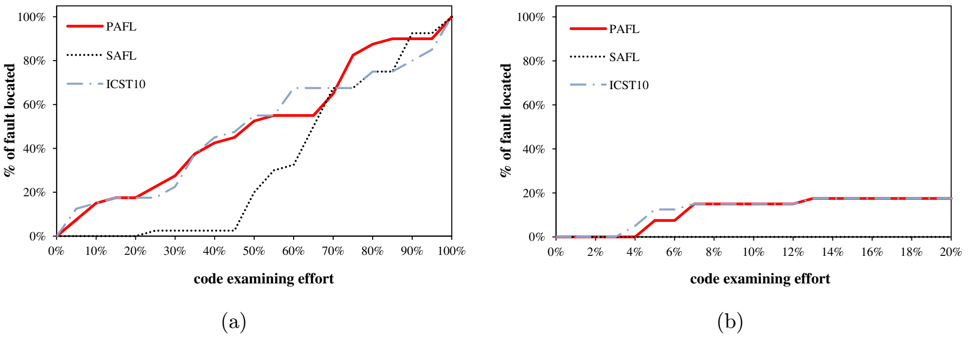
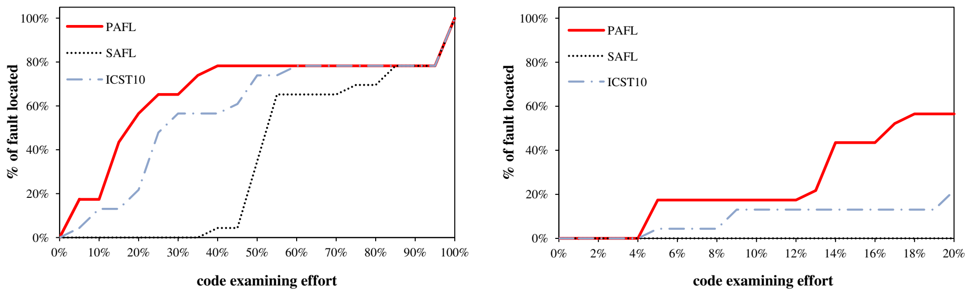
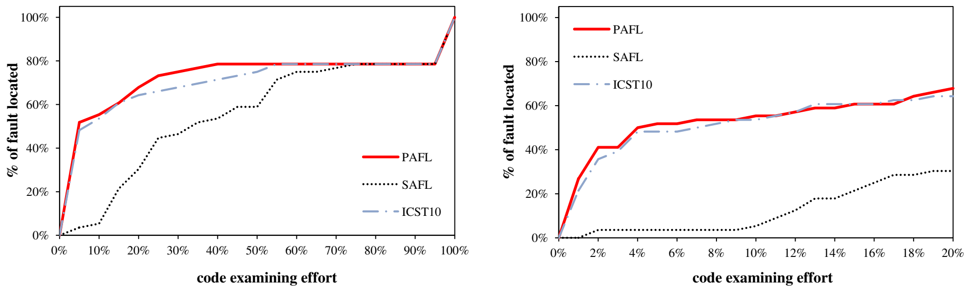


Fig. 12. Results on *tcas* in (a) full range [0%, 100%] and (b) zoom-in range of [0%, 20%].



(a) (b)

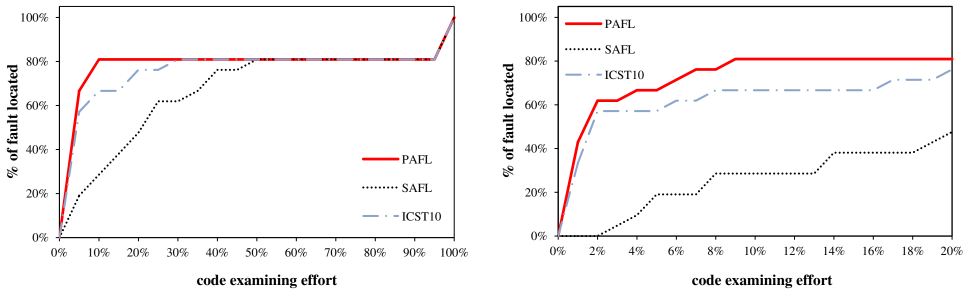
Fig. 13. Results on *tot_info* in (a) full range [0%, 100%] and (b) zoom-in range of [0%, 20%].



(a) (b)

Fig. 14. Results on *flex* in (a) full range [0%, 100%] and (b) zoom-in range of [0%, 20%].

respectively. Also, PAFL outperforms the other two in the best case and at the 25% percentile measurement. An exception is that at the 75% percentile measurement ICST10 catches up with PAFL (the code examining efforts for PAFL and ICST10 at the 75% percentile are 61.5% and 59.5%, respectively).



(a) (b)

Fig. 15. Results on *grep* in (a) full range [0%, 100%] and (b) zoom-in range of [0%, 20%].

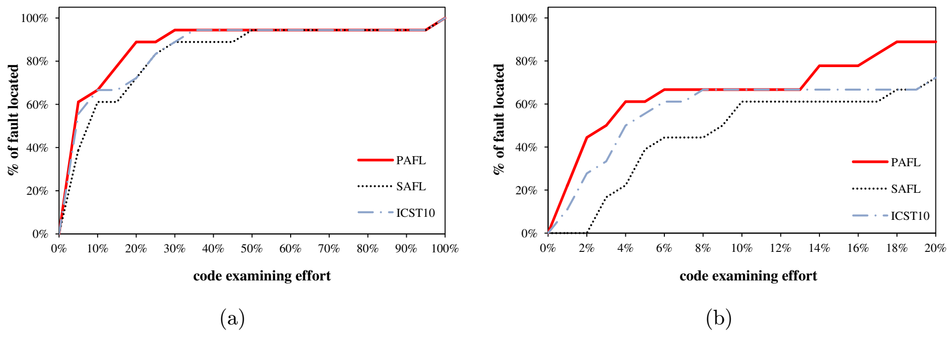


Fig. 16. Results on *gzip* in (a) full range [0%, 100%] and (b) zoom-in range of [0%, 20%].

Till now, we have compared the overall effectiveness of the techniques studied. We are also interested in the effectiveness on each individual program.

Figures 7 to 16 show the results on the ten programs, both in the full range and the [0%, 20%] zoomed-in range. From Figs. 7 to 16, we observe that in the full range, PAFL is more effective than the other techniques in the plots of *print_tokens*, *print_tokens*2, and *tot_info*, *flex*, *grep*, and *gzip*. In the plots of *replace*, *schedule*, *schedule*2, and *tcas*, PAFL is less effective than some of other techniques in some regions.

We also use Table 4 to assemble the mean effectiveness of these techniques on each different program. The mean effectiveness is Table 4 refers to the average code examination percentage to locate faults. That is, the technique is more effectiveness with lower mean effectiveness. This table shows that for these programs, PAFL is always (except on *schedule*) the best among the three techniques. For example, PAFL takes on average 16.8% code examining effort to locate a fault in the faulty versions of program *print_tokens*2, while SAFL and ICST10 need to examine 54.9% and 23.6% code, respectively. From the individual results on each program, we have also the consistent observation that the most effective technique synthesized in our framework is mostly more effective, or at least as effective as, the two peer techniques SAFL and ICST10.

Table 4. Mean effectiveness on individual programs.

Types	Subjects	PAFL	SAFL	ICST10
Siemens	print_tokens	**72.5%**	84.2%	73.7%
	print_tokens2	**16.8%**	54.9%	23.6%
	replace	**19.6%**	36.6%	20.8%
	schedule	25.1%	52.8%	**24.3%**
	schedule2	**81.5%**	82.5%	83.8%
	tcas	**50.3%**	66.5%	51.5%
	tot_info	**34.3%**	64.3%	42.9%
UNIX	flex	**27.2%**	44.5%	29.5%
	grep	**20.6%**	34.1%	23.2%
	gzip	**11.6%**	17.6%	14.3%

Table 5. Expenses of different techniques on the multi-fault version of *replace*.

	PAFL	Ochiai	Jaccard	Tarantula	SBI	ICST10	SAFL
Minimum	16.4	17.2%	17.2%	**12.7%**	**12.7%**	**12.7%**	18.4%
25% percentile	**41.8%**	46.3%	46.2%	50.2%	50.2%	50.2%	48.2%
Median	**63.5**%	70.5%	68.9%	77.5%	77.5%	77.5%	80.5%
75% percentile	89.2%	90.6%	90.0%	**85.9**%	**85.9**%	**85.9**%	93.1%
Average	**64.4**%	66.8%	66.4%	67.5%	67.5%	67.5%	68.5%
Maximum	100%	100%	100%	100%	100%	100%	100%

4.8. *A case study on multi-fault programs*

In this section, we construct a case study to validate the effectiveness of PAFL on multi-fault programs. Since the test suites of programs in Siemens contain more similar test cases, we select from Siemens and make *replace* as the subject program. *replace* has 29 faulty versions and we try to inject each fault into the multi-fault version of *replace*. Note that, the faults in some faulty versions could exist in the same statements or expressions, in such cases, we only inject the fault with lower version number. After all, we inject 14 faults into the multi-fault version of *replace*.

To evaluate the effectiveness of PAFL, we employ different CBFL techniques to work on the multi-fault version of *replace*, and then compare the expense of techniques to locate every faults. The results are shown in Table 5.

As shown in Table 5, we list the statistical of different techniques, which include the minimum values of code examining efforts to locate the 14 faults, the 25% percentile values, the median values, the 75% percentile values, the average values and the maximum values. We observe that the results of PAFL are better than other techniques, expect the minimum and 75% percentile value. To some extent, it shows that PAFL could also perform better on multi-fault programs.

4.9. *Threats to validity*

We use the UNIX tool *gcov* to collect the execution spectra and generate coverage vectors. The limitation of the tool *gcov* prevents the crashed program executions from being processed, and thus causes incomplete data collection. Powerful instrumentation tools are expected to improve the effectiveness of related techniques.

The coverage vectors are used to count the distinct paths, but vectors and paths are not equal. If two coverage vectors are identical, the corresponding paths may be still different. We choose to use coverage vector to capture the path information because only in such a way can we fairly compare our approach with a popular family of fault localization techniques, which also use coverage information as input.

We use seven Siemens programs and three median-sized real-life UNIX utilities as subject programs to evaluate the studied techniques. Though these programs are extensively used in previous studies, using other large-sized programs or programs containing real faults may also be good choices.

We download test pool from SIR for each program. Since those test cases are generated to satisfy different coverage criteria, such a generation strategy somehow implies that there is less execution similarity. Adopting some other test case generation strategies may affect the experiment results.

Effectiveness metrics also causes threat to the validity of the experiment. Different from some of previous work, in our experiment, we calculate the code examining effort as the percentage of basic blocks examined before reaching a fault. The reason we do so has been explained in Sec. 4.3. When there exist some basic blocks that form a tie case and cannot be break, our strategy is to examine them as a whole and count the code examining effort. Adopting other strategies (e.g., examine them in their appearance order in the program or suppose the best case scenario where the faulty basic block is examined instead of the whole tie) may generate different results.

5. Related Work

In this section, we review related work on fault localization research.

5.1. *Program slicing*

Program slicing is first proposed by Weiser [57], and these techniques include static [57] and dynamic slicing [12, 58]. Slicing techniques, especially dynamic slicing, are widely used to support software debugging [59, 13, 60].

A slice refers to a set of statements in a program that may affect the computed values at some location. In general, by slicing statements which are related to the observed failure variables or failure inducing inputs, programmers can filter out statements for fault localization and debugging. Static slicing techniques analyze the semantic relationship with observed variables, but it need to explorer the whole program states space, which may lead to the heavyweight and time-consuming computation. To address this problem, lots of approaches have been proposed to optimize the computation. For example, Ottenstein [61] develops a program dependence graph (PDG) to reduce the computation of static slices of a sequential program to a reach ability problem in PDG. Horwitz *et al.* [62] extend the technique to inter-procedural slicing. However, previous studies also show that the size of a static slice for a program can be one-third of the program [14]. During program debugging or fault localization, it is still very hard for programmers to inspect and look for faulty statements.

For a dynamic execution of the program with a given input, dynamic slicing focuses on the statements which are executed in the program run, and through execution indexing, dynamic slicing can further record the dynamic dependency. By doing these, dynamic slicing can reduce the size of sliced statements, which are widely used to support program debugging in recent years. Chen and Cheung [63] propose dynamic dicing and the related strategies to construct dynamic dices. Gupta *et al.* [13] propose to use both forward and backward dynamic slicing to narrow down slices. Zhang *et al.* [60] propose to prune statements with confidence by slicing statements according different observed program outputs. By comparing the failed execution with a passed execution, Johnson *et al.* [64] propose a fine-grained slicing approach, differential slicing. Slicing techniques analyze both the failed outputs and failure inducing inputs, whereas CBFL techniques mainly focus the execution spectra of executions. These two kinds of techniques can be integrated to improve the effectiveness of fault localization and fault fixing assistant. Since they use different information as inputs, we do not compare them in this paper.

5.2. *Coverage based fault localization techniques*

Agrawal *et al.* [65] propose the coverage based fault localization technique χSlice. By comparing the execution spectra between two executions (one is passed and the other is failed), this approach filters out the statements which are only exercised in the failed executions for further inspection. This idea is further developed by Renieris and Reiss [24], who propose the Nearest Neighborhood (NN) technique. Using statistical approach, Jones and Harrold [15, 8] propose Tarantula to rank every statement according to its fault suspiciousness. Abreu *et al.* [22] adopt two statistical approaches to propose the Ochiai and Jaccard techniques, which are used for comparison in our paper. Abreu *et al.* [16] further show empirically that a technique can achieve almost the same fault localization accuracy by using a few failed executions.

Several other approaches use statistical measures for behaviors related to program failures. Jones *et al.* [66] further extend Tarantula so that it can be applied when multiple developers are available to debug the program independently. Observing that individual executions of the same statement may have different contributions to indicate faulty statements when they are used together, Wong *et al.* [48] propose to use a utility function to calibrate the contribution of each passed execution when computing the fault relevance of executed statements. This work proposes that the contribution of all the successful test cases to program debugging should not be treated equally, of which the basic idea inspired our work. However, our framework uses coverage vector to estimate the execution similarity whereas Wong *et al.* [48] uses the execution spectra of statements. They further define a series of heuristics based on different marginal contributions of additional failed executions and passed executions [67]. Debroy and Wong [68] propose a cross-tab method to compute the fault suspiciousness of statements and focus on programs having multiple faults.

Harrold *et al.* [69] evaluate nine kinds of program features, including path counts, data-dependency counts, and execution traces. Later studies show, however, that by applying a proper contrast step, the usage of data-dependency counts can be more effective than that of control-dependency counts [70]. CBI [3, 20] and SOBER [56, 17] are two representative techniques that relate to control-dependency information. More specifically, they make use of the execution spectra information of program predicates set in branch statements and so on, and hence we call them *predicate based CBFL techniques*. CBI compares the probability that a program fails when a predicate is ever evaluated to be true with the probability that the program fails when the predicate is ever evaluated. The technique uses this difference as the primary program feature to identify the positions of the predicates related to faults. SOBER further proposes to use the actual probability that a predicate is evaluated to be true, which they call the evaluation bias, as the program feature. It contrasts the evaluation biases of each predicate in passed and failed executions to locate predicates that are related to faults. After locating suspicious predicates, these methods recommend programmers to search for faults around the located suspicious predicates in the program. In this paper, we compare with four representative of these techniques.

5.3. *Other fault localization techniques*

Delta debugging [9, 11] simplifies the failed test cases and yet preserves the failures, producing cause-effect chains and linking them to suspicious statements.

Arumuga Nainar *et al.* [71] further extend CBI to address compound Boolean expressions. They show that the accuracy of CBI changes significantly when compound Boolean expressions are involved. Zhang *et al.* [5, 72] conduct an empirical study to show that the short-circuit rules in evaluating boolean expressions in predicates affect the effectiveness of fault-localization techniques, and that the results of CBI can be improved using evaluation sequences information in the form of evaluation sequences. Chilimbi *et al.* [73] propose HOLMES, which uses fragments of paths rather than individual predicates to locate faults iteratively. Zhang *et al.* [74] find empirically that the evaluation biases of many predicates are not distributed normally. They further [19] propose a generic framework of predicate based fault localization techniques, apply many non-parametric, parametric, and debugging specific hypothesis testing methods to generate predicate-based fault localization techniques, and empirically evaluate them.

Based on the suspiciousness estimation obtained from a contrast step, CP [4] constructs a probabilistic control flow graph and a propagation model for the faulty program with a view to capturing the propagation of infected states extracted from the given set of program executions to locate faults.

Jiang and Su [75] use clustering to obtain fault predictors with the biggest fault proneness and generate the execution paths traversing these predicates to reflect how the failure occurs. Baah *et al.* [76] presents an innovative model of a program's internal behavior over a set of test inputs, called the probabilistic program dependence graph (PPDG), that facilitates probabilistic analysis and reasoning about uncertain program behavior, particularly that associated with faults. Many other methodologies, such as training a neural network [23]. However, comparing with these techniques exceeds the scope of this paper.

Hao *et al.* [6] propose a similarity-aware fault localization technique SAFL. In SAFL, the similarity of test cases is evaluated based the fuzzy set and the coverage intersection of test cases. However, it is not always stand to make all executions that have coverage intersection as similar, because executions are generally have more or less coverage intersections. Compared with SAFL, the similarity evaluation is much stricter in PAFL.

5.4. *Integrating fault localization, test cases generation, and test suite reduction*

There are also some studies focusing on the test suite reduction to enhance fault localization, which are related to our work.

Some techniques aim to minimum the similar test cases to satisfy some coverage criteria, while our framework aims to alleviate the impact of execution similarity when locating faults. Hao *et al.* [27] focus on optimizing the size of test inputs to facilitate effective fault localization. Jiang *et al.* [55] point out that test suites prioritized by coverage-based strategies are better than those from other strategies in terms of the effectiveness of fault localization. Artzi *et al.* [77] adopt the dynamic symbolic execution technique to generate test cases for statistical fault localization. Their study shows that the path-constraint based technique can generate the smallest test suites with the same excellent fault-localization characteristics as test suites generated by other techniques. Baudry *et al.* [78] observe that some statements are always executed by the same set of executions. They propose the dynamic basic blocks and use the execution spectra of dynamic basic blocks to improve the test suite, empirical studies show that their approach requires fewer test cases to achieve the same fault localization effectiveness. These studies either focus on the test suite reduction for effective fault localization, or how to generate test cases for statistical fault localization. In this paper, our framework aims to improve the effectiveness of fault localization if the test suite is not optimized and there may be similar test cases in the test suite.

In [21], Yu *et al.* propose test cases reduction approaches based on coverage vectors, which is similar with the similarity evaluation in PAFL. The difference is the program scope, and PAFL further evaluates the failing extent of coverage vectors for tie-braking.

6. Conclusion

In this paper, we demonstrate that frequently occurred execution similarity may affect the effectiveness of existing techniques and propose a general framework to synthesize new fault localization techniques from base ones, to address the problem of execution similarity. In our framework, we first use the concept of coverage vectors to count distinct execution paths and model execution spectra, compare coverage vectors to capture the execution similarity. We then calculate the failing extent of each distinct coverage vector, and mark it as a failed or passed distinct coverage vector according to whether it is ever exercised in a failed execution or not. Next, for each basic block, we evaluate the suspiciousness score of that basic block, using the execution statistics of distinct coverage vectors. At last, we sort all the basic blocks in the descending order of their computed suspiciousness scores.

We adopt four representative fault localization techniques as base techniques, use seven Siemens programs and three median-sized real-life UNIX utility programs as subject programs, to conduct an experimental study on the effectiveness of our framework. The empirical evaluation shows that our framework can alleviate the impact of execution similarity, and synthesize more effective fault localization techniques based on existing ones.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Nos. 61303213), the research fund of the State Key Laboratory for Novel Software Technology, and the research fund of the Key Laboratory for Information Assurance Technology (KJ-13).

References

1. J. Duraes and H. Madeira, Emulation of software faults: A field data study and a practical approach, *IEEE Transactions on Software Engineering* **32**(11) (2006) 849–867.
2. H. Agrawal, R. A. DeMillo and E. H. Spafford, An execution backtracking approach to program debugging, *IEEE Software* **8**(3) (1991) 21–26.
3. B. Liblit, A. Aiken, A. Zheng and M. Jordan, Bug isolation via remote program sampling, in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI 2003)*, ACM, 2003, pp. 141–154.
4. Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang and X. Wang, Capturing propagation of infected program states, in *Proceedings of the 17th International Conference on Foundation of Software Engineering (FSE/ESEC 2009)*, ACM, 2009, pp. 43–52.
5. Z. Zhang, B. Jiang, W. K. Chan and T. H. Tse, Debugging through evaluation sequences: A controlled experimental study, in *Proceedings of the 32nd Annual International Computer Software and Applications Conference (COMPSAC 2008)*, IEEE Computer Society, 2008, pp. 128–135.
6. D. Hao, L. Zhang, Y. Pan, H. Mei and J. Sun, On similarity-awareness in testing-based fault localization, *Journal of Automated Software Engineering* **2008**(15) (2008) 207–249.
7. I. Vessey, Expertise in debugging computer programs: An analysis of the content of verbal protocols, *IEEE Transactions on Systems, Man, and Cybernetics* **16**(5) (1986) 621–637.
8. J. A. Jones and M. J. Harrold, Empirical evaluation of the tarantula automatic fault-localization technique, in *Proceedings of the 20th IEEE International Conference on Automated Software Engineering (ASE 2005)*, ACM, 2005, pp. 273–282.
9. A. Zeller, Isolating causec-effect chains from computer programs, in *Proceedings of the 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT 2002/FSE-10)*, ACM, 2002, pp. 1–10.
10. A. Zeller and R. Hildebrandt, Simplifying and isolating failure-inducing input, *IEEE Transactions on Software Engineering* **28**(2) (2002) 183–200.
11. H. Cleve and A. Zeller, Locating causes of program failures, in *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, ACM, 2005, pp. 342–351.

12. H. Agrawal and J. R. Horgan, Dynamic program slicing, in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI 1990)*, ACM, 1990, pp. 246–256.
13. N. Gupta, H. He, X. Zhang and R. Gupta, Locating faulty code using failure inducing chops, in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, ACM, 2005, pp. 263–272.
14. D. Binkley, N. Gold and M. Harman, An emprical study of static program slice size, *ACM Transactions on Software Engineering and Methodology* **16**(2) (2007) 8–es.
15. J. A. Jones and M. J. Harrold, Visualization of test information to assist fault localization, in *Proceedings of the International Conference on Software Engineering (ICSE 2002)*, ACM, 2002, pp. 467–477.
16. R. Abreu, P. Zoeteweij, R. Golsteijn and A. J. van Gemund, A practical evaluation of spectrum-based fault localization, *Journal of Systems and Software* **82**(11) (2009) 1780–1792.
17. C. Liu, L. Fei, X. F. Yan, J. W. Han and S. Midkiff, Statistical debugging: A hypothesis testing-based approach, *IEEE Transaction on Software Engineering* **32**(10) (2006) 1–17.
18. L. Naish, H. Lee and K. Ramamohanarao, A model for spectra-based software diagnosis, *ACM Transactions on Software Engineering and Maintenance* **20**(3) (2011) 11–es.
19. Z. Zhang, W. K. Chan, T. H. Tse, Y. T. Yu and P. Hu, Non-parametric statistical fault localization, *Journal of Systems and Software* **84**(6) (2011) 885–905.
20. B. Liblit, M. Naik, A. X. Zheng, A. Aiken and M. I. Jordan, Scalable statistical bug isolation, in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)*, ACM, 2005, pp. 15–26.
21. Y. Yu, J. A. Jones and M. J. Harrold, An empirical study of the effects of test-suite reduction on fault localization, in *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, ACM, 2008, pp. 201–210.
22. R. Abreu, P. Zoeteweij and A. J. van Gemund, On the accuracy of spectrum-based fault localization, in *Proceedings of the Testing: Academic and Industrial Conference, Practice and Research Techniques*, 2007, pp. 89–98.
23. W. E. Wong and Y. Qi, Bp neural network-based effective fault localization, *Journal of Software Engineering and Knowledge Engineering* **19**(4) (2009) 573–597.
24. M. Renieri and S. Reiss, Fault localization with nearest neighbor queries, in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, ACM, 2003, pp. 30–39.
25. G. J. Myers, *The Art of Software Testing* (John Wiley & Sons, New York, 1976).
26. S. G. Elbaum, A. G. Malishevsky and G. Rothermel, Test case prioritization: A family of empirical studies, *IEEE Transactions on Software Engineering* **28**(2) (2002) 159–182.
27. D. Hao, T. Xie, L. Zhang, X. Wang, J. Sun and H. Mei, Test input reduction for result inspection to facilitate fault localization, *Automated Software Engineering* **17**(1) (2010) 5–31.
28. R. Hierons, Avoiding coincidental correctness in boundary value analysis, *ACM Transaction of Software Engineering and Maintenance* **15**(3) (2006) 227–241.
29. X. Wang, S. C. Cheung, W. K. Chan and Z. Zhang, Taming coincidental correctness: Refine code coverage with context pattern to improve fault localization, in *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, ACM, 2009, pp. 45–55.
30. L. Zhao, Z. Zhang, L. Wang and X. Yin, Pafl: Fault localization via noise reduction on coverage vector, in *Proceedings of the 23rd International Conference on Software Engineering and Knowledge Engineering (SEKE 2011)*, 2011, pp. 203–206.

31. W. Masri and R. A. Assi, Cleansing test suites from coincidental correctness to enhance fault-localization, in *Proceedings of the 2010 International Conference on Software Testing*, IEEE Computer Society, 2010, pp. 165–174.
32. J. Xu, W. K. Chan, Z. Zhang and T. H. Tse, A dynamic fault localization technique with noise reduction for Java programs, in *Proceedings of the 11th International Conference on Quality Software (QSIC 2011)*, IEEE Computer Society, 2011, pp. 11–20.
33. P. Jaccard, Etude comparative de la distribution florale dans une portion des Alpes et du Jura, *Bulletin del la Socit Vaudoise des Sciences Naturelles* **1901**(37) 547–579.
34. M. R. Anderberg, *Cluster Analysis for Applications* (Academic Press, 1973).
35. J. Duarte and J. Santos, Comparison of similarity coefficients based on RAPD markers in the common bean, *Genetics and Molecular Biology* **22**(3) (1999).
36. L. Dice, Measures of the amount of ecologic association between species, *Ecology* **26**(3) (1945) 297–302.
37. F. Lourenco, V. Lobo and F. Bacao, Binary-based similarity measures for categorical data and their application in self-organizing maps, *Procs. JOCLAD*, 2004.
38. P. F. Russell, On habitat and association of species of anopheline larvae in south-eastern Madras, *Journal of the Malaria Institute of India* **3** (1940) 153–178.
39. A. d. S. Meyer and A. Garcia, Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (zea mays l), *Genetics and Molecular Biology* **27** (2004) 83–91.
40. B. Everitt, *Graphical Techniques for Multivariate Data* (North-Holland, 1978).
41. D. J. Rogers and T. T. Tanimoto, A computer program for classifying plants, *Science* **132**(3434) (1960) 1115–1118.
42. L. Goodman, Measures of association for cross classifications III: Approximate sampling theory, *Journal of the American Statistical Association* **49** (1963) 732–764.
43. R. Hamming, Error detecting and error correcting codes, *Bell System Technical Journal* **29**(1) (1950) 147–160.
44. E. Krause, Taxicab geometry, *Mathematics Teacher* **66**(8) (1973) 695–706.
45. A. Ochiai, Zoogeographic studies on the soleoid fishes found in Japan and its neighbouring regions, *Bull. Jpn. Soc. Sci. Fish* **22** (1957) 526–530.
46. A. Gonzalez, Automatic error detection techniques based on dynamic invariants, Master's Thesis, Delft University of Technology, Delft, Netherlands, 2007.
47. V. Dallmeier and C. Lindig, Lightweight bug localization with AMPLE, in *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging (AADEBUG 2005)*, ACM, 2005, pp. 99–104.
48. W. E. Wong, Y. Qi, L. Zhao and K. Cai, Effective fault localization using code coverage, in *Proceedings of the 31st Annual International Computer Software and Application Conference (COMPSAC 2007)*, IEEE Computer Society, 2007, pp. 449–456.
49. A. Maxwell and A. Pilliner, Deriving coefficients of reliability and agreement for ratings, *British Journal of Mathematical and Statistical Psychology* **21**(1) (1968) 105–116.
50. E. Rogot and I. D. Goldberg, A proposed index for measuring agreement in test-retest studies, *Journal of Chronic Diseases* **19**(9) (1966) 991–1006.
51. J. Cohen, A coefficient of agreement for nominal scales, *Educational and Psychological Measurement* **20**(1) (1960) 37–46.
52. W. A. Scott, Reliability of content analysis: The case of nominal scale coding, *Public Opinion Quarterly* **19**(3) (1955) 321–325.
53. J. L. Fleiss, Estimating the accuracy of dichotomous judgments, *Psychometrika* **30**(4) (1965) 469–479.

54. H. Do, S. Elbaum and G. Rothermel, Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact, *Empirical Software Engineering* **10**(4) (2005) 405–435.
55. B. Jiang, Z. Zhang, T. H. Tse and T. Y. Chen, How well do test case prioritization techniques support statistical fault localization, in *Proceedings of the 33rd Annual International Computer Software and Applications Conference (COMPSAC 2009)*, IEEE Computer Society Press, 2009, pp. 99–106.
56. C. Liu, X. Yan, L. Fei, J. Han and S. P. Midkiff, Sober: Statistical model-based bug localization, in *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC 2005/FSE-13)*, ACM, 2005, pp. 286–295.
57. M. Weiser, Program slicing, *IEEE Transactions on Software Engineering* **10**(4) (1984) 352–357.
58. B. Korel and J. Laski, Dynamic program slicing, *Information Processing Letters* **29**(3) (1988) 155–163.
59. F. Tip, A survey of program slicing techniques, *Journal of Programming Languages* **3**(3) (1995) 121–189.
60. X. Zhang, N. Gupta and R. Gupta, Pruning dynamic slices with confidence, in *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI 2006)*, ACM, 2006, pp. 169–180.
61. K. J. Ottenstein and L. M. Ottenstein, The program dependence graph in a software development environment, in *Proceedings of the ACM Symposium on Practical Software Development Environments*, ACM, 1984, pp. 177–184.
62. S. Horwitz, T. Reps and D. Binkley, Interprocedural slicing using dependence graphs, *ACM Transactions on Programming Languages and Systems* **12**(1) (1990) 26–60.
63. T. Y. Chen and Y. Y. Cheung, Dynamic program dicing, in *Proceedings of the 9th IEEE International Conference on Software Maintenance (ICSM 1993)*, ACM, 1993, pp. 378–385.
64. N. Johnson, J. Caballero, K. Z. Chen, S. McCamant, P. Poosankam, D. Reynaud and D. Song, Differential slicing: Identifying causal execution differences for security applications, in *Proceedings of the 32nd IEEE Symposium on Security and Privacy*, ACM, 2011, pp. 347–362.
65. H. Agrawal, J. Horgan, S. Lodon and W. Wong, Fault localization using execution slices and dataflow tests, in *Proceedings of the 6th International Symposium on Software Reliability Engineering*, IEEE Computer Society, 1995, pp. 143–151.
66. J. A. Jones, J. F. Bowring and M. J. Harrold, Debugging in parallel, in *Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007)*, ACM, 2007, pp. 16–26.
67. W. E. Wong, V. Debroy and B. Choi, A family of code coverage-based heuristics for effective fault localization, *Journal of Systems and Software* **83**(2) (2010) 188–208.
68. V. Debroy and W. E. Wong, Insights on fault interference for programs with multiple bugs, in *Proceedings of the 20th International Symposium on Software Reliability Engineering (ISSRE 2009)*, IEEE Computer Society, 2009, pp. 165–174.
69. 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* **10**(3) (2000) 171–194.
70. R. Santelices, J. A. Jones, Y. Yu and M. J. Harrold, Lightweight fault localization using multiple coverage types, in *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, ACM, 2009, pp. 56–66.

71. P. Arumuga Nainar, T. Chen, J. Rosin and B. Liblit, Statistical debugging using compound boolean predicates, in *Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2007)*, ACM, 2007, pp. 5–15.
72. Z. Zhang, B. Jiang, W. K. Chan, T. H. Tse and X. Wang, Fault localization through evaluation sequences, *Journal of Systems and Software* **83**(2) (2010) 174–187.
73. T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori and K. Vaswani, Holmes: Effective statistical debugging via efficient path profiling, in *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009)*, ACM, 2009, pp. 34–44.
74. Z. Zhang, W. K. Chan, T. H. Tse, P. Hu and X. Wang, Is non-parametric hypothesis testing model robust for statistical fault localization? *Information and Software Technology* **51**(11) (2009) 1573–1585.
75. L. Jiang and Z. Su, Context-aware statistical debugging: From bug predictors to faulty control flow paths, in *Proceedings of the 22nd IEEE International Conference on Automated Software Engineering (ASE 2007)*, ACM, 2007, pp. 184–193.
76. G. K. Baah, A. Podgurski and M. J. Harrold, The probabilistic program dependence graph and its application to fault diagnosis, in *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA 2008)*, ACM, 2008, pp. 189–200.
77. S. Artzi, J. Dolby, F. Tip and M. Pistoia, Directed test generation for effective fault localization, in *Proceedings of the 2010 Internet Symposium on Software Testing and Analysis (ISSTA 2010)*, ACM, 2010, pp. 49–59.
78. B. Baudry, F. Fleurey and Y. Le Traon, Improving test suites for efficient fault localization, in *Proceedings of the 26th International Conference on Software Engineering (ICSE 2006)*, ACM, 2006, pp. 82–91.