PAFL: Fault Localization via Noise Reduction on Coverage Vector

Lei Zhao
Computer School of
Wuhan University
Wuhan, China, 430072
zhaolei.whu@gmail.com

Zhenyu Zhang
State Key Laboratory of Computer Science,
Institute of Software, Chinese Academy of Sciences
Beijing, China, 100190
zhangzy@ios.ac.cn

Lina Wang
Computer School of Wuhan University, Key Laboratory of
Aerospace Information Security and Trust Computing
Wuhan, China, 430072
lnwang@whu.edu.cn

Xiaodan Yin
Computer School of
Wuhan University
Wuhan, China, 430072
yinxiaodan.whu@gmail.com

***Abstract*—Coverage-based fault localization techniques assess the extent of how much a program entity relates to faults by contrasting the execution spectra of passed executions and failed executions. However, previous studies show that different test cases may generate similar or identical coverage information in program execution, which makes the execution spectra of program entities indistinguishable to one another, thus involves noise and decreases the effectiveness of existing techniques. In this paper, we use the concept of coverage vector to model program coverage in execution, compare coverage vectors to capture the similarity among test cases, reduce noise by removing similar coverage vector to refine the execution spectra, and based on them assess the suspicious basic blocks being related to fault. We thus narrow down the search region and facilitate fault localization. The empirical evaluation using Siemens programs and realistic UNIX utilities shows that our technique effectively addresses the problem caused by similar test cases and outperforms existing representative techniques.**

Keywords-fault localization; execution path; noise reduction

I. INTRODUCTION

Coverage-based fault localization (CBFL) techniques have been proposed to support software debugging [8][11] [13]. By contrasting the coverage statistics of program entities (such as statements, blocks and predicates) between passed executions and failed executions, CBFL techniques can locate the program entities which exercising are strongly correlated to the program execution failures observed. Previous studies showed that CBFL techniques are effective in locating faults [8][13].

Since 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, it is common that different executions may cover similar and even identical execution paths [4]. Similar coverage information makes the execution spectra of program entities indistinguishable in passed and failed executions and thus decreases the effectiveness of previous fault localization techniques or even makes them lose effect, especially when coincidental correctness occurs [5]. For example, suppose a faulty statement is exercised in the program execution of all passed and all failed executions, it is hard to pinpoint it by contrasting its execution spectra in passed and failed executions. Such a case may have a high chance to happen in real life programs (e.g., a faulty statement may exist in the main method of a program and must be exercised by all the executions). Previous study also shows that execution similarity and coincidental correctness occurs frequently in realistic programs [12].

In this paper, we propose to use the concept of coverage vector to count the distinct execution paths, to capture the happening of execution similarity. We first calculate the failing rate of each coverage vector as the ratio of the number of failed executions covering it to the number of all executions covering it. For each basic block, we then calculate two numbers, the number of coverage vectors exercising that basic block and covered by failed executions, and the number of coverage vectors either exercising that basic block or covered by failed executions. We then use the ratio of the former to the latter as the suspiciousness score of that basic block. We next sort all the basic blocks in the descending order of thus calculated suspiciousness scores. In case of a tie, which means two or more blocks sharing identical suspiciousness scores, we try to use the average failing rate of coverage vectors exercising a block to further determine the order of the blocks.

We use seven Siemens programs and three UNIX utilities to evaluate our technique, and compare it with five representative techniques, namely, Tarantula [7], Jaccard [1], SBI [13], SAFL [4], and ICST10 [10]. The empirical results show that our technique is promising on the studies subject programs. Further analysis show that our technique is promising to alleviate the impact of execution similarity.

The contributions of this paper are twofold. (i) We propose to use the concept of coverage vector to count the distinct execution paths, and make use of it to estimate the occurring of similarities from the coverage information. (ii) We propose a new fault localization technique, PAFL, which is empirically evaluated to be promising in locating fault, especially in case of high execution similarity.

II. MOTIVATING EXAMPLE

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

The code excerpt in Figure 1 finds the middle value in three given numbers. A fault exists in statement s_2, which accesses the variable *x* instead of *z*. We choose six test cases to demonstrate in this example. The mark "•" in each cell indicates that a statement is exercised in the program

Blocks	Statements	Test cases						Previous techniques										Distinct paths				Our approach	
		t_1	t_2	t_3	t_4	t_5	t_6	Tarantula		Jaccard		SBI		SAFL		ICST10		p_1		p_2			
		F	*P*	*P*	*P*	***F***	*P*	score	rank	score	rank	score	rank	score	rank	score	rank	t_1	t_2,t_3,t_4	t_5	t_6	score	rank
	Mid() {																						
	int x, y, z, m;																						
	s_1 read ("Enter 3 num:", x, y, z);																						
b_1	s_2 m=x; **/* a mutant of m=z */**	•	•	•	•	•	•	0.50	4	0.33	4	0.50	4	0.81	6	0.50	4	•	•	•	•	0.50	2
	s_3 if (y<z)																						
b_2	s_4 if (x<y)					•	•	0.67	2	0.33	4	0.67	2	1	5	0.67	2			•	•	0.33	6
b_3	s_5 m=y;																						
b_4	s_6 else if (x<z)					•	•	0.67	2	0.33	4	0.67	2	1	5	0.67	2			•	•	0.33	6
b_5	s_7 m=x;																						
b_6	s_8 else if (x>y)	•	•	•	•			0.40	6	0.20	6	0.40	6	1	5	0.40	6	•	•			0.33	6
b_7	s_9 m=y;																						
b_8	s_{10} else if (x>z)	•	•	•	•			0.40	6	0.20	6	0.40	6	1	5	0.40	6	•	•			0.33	6
b_9	s_{11} m=x;																						
b_{10}	s_{12} printf ("The middle is:", m);	•	•	•	•	•	•	0.50	4	0.33	4	0.50	4	1	5	0.50	4	•	•	•	•	0.50	2
	}																						
	Code examining effort to locate fault:							66%		66%		66%		100%		66%						33%	

Figure 1. Motivating example – the program Mid

execution with respect to a test case. The execution results (*P* for pass and *F* for fail) are shown in table header.

We apply previous techniques Tarantula [7], Jaccard [1], SBI [13], SAFL [4], and ICST10 [10] to locate fault (statement s_2) in this example. For example, Tarantula assigns suspiciousness score 0.50 to statement s_2, and finally needs to examine 66% of all code to locate the fault. The results of Jaccard, SBI, SAFL, and ICST10 can be similarly explained. Unfortunately, none of them can locate the fault with somehow affordable code examining efforts (e.g., less than 50%). This is because that the faulty statement (s_2) happen to be exercised in all passed and failed executions, so that they are indistinguishable in execution spectra. On the other hand, Statements s_4 and s_6 are given higher suspiciousness scores than statements s_8 and s_{10} because the former happen to be exercised in relatively more failed executions than the latter. Note that statements s_4 and s_6 are exercised in one failed execution (t_5) and one passed execution (t_6), while statements s_8 and s_{10} are exercised in one failed execution (t_1) and three passed executions (t_2, t_3, and t_4). Among the two passed test cases (t_5 and t_6), which exercise the former (statements s_4 and s_6), 50% of them are failed ones. While only 25% of the test cases that exercise the latter (statements s_8 and s_{10}) are failed ones. As a result, statements s_4 and s_6 are given higher suspiciousness scores because of such imbalance.

Execution similarity may frequently occur in realistic programs [12]. We could not expect to always benefit from the imbalance observed in previous paragraph. To measure the execution similarity, we design to use the coverage information. In addition, we also measure the execution similarity to estimate the occurring of coincidental correctness in passed executions and manage to alleviate the impact of coincidental correctness to execution spectra of statements.

From Figure 1, we observe that there are in total two distinct paths covered by the six test cases [2], which are denoted as $p_1 = \langle b_1, b_2, b_4, b_{10}\rangle$ and $p_2 = \langle b_1, b_3, b_7, b_{10}\rangle$. After that, we adopt SBI's formula $\frac{\text{failed(p)}}{\text{failed(p)+passed(p)}}$ to estimate the failing rate of a path, which means the probability of an execution (covering that path) revealing a failure. Note that, here we use a path, rather than a statement, as the program entity in the formula. Since p_1 is covered by one failed test case and three passed test cases, the failing rate of p_1 is 0.25. Similarly, the failing rate of p_2 is 0.5. A path with a positive failing rate means that the corresponding execution can reveal failures. A path with failing rate zero means that the corresponding execution reveals no failure. We then adopt Jaccard's formula $\frac{\text{failed(b)}}{\text{totalfailed+passed(b)}}$ to calculate the suspiciousness score for each block. Here, *failed*(*b*) means the number of distinct paths, which have positive failing rates and exercise block *b*; *passed*(*b*) means the number of distinct paths, which have failing rates of zero and exercise block *b*; *totalfailed* means the number of distinct paths, which have positive failing rates. Thus, the suspiciousness score of block b_1 is calculated as $\frac{2}{2+2} = 0.50$. As a result, we finally take 33% code examining effort to locate the fault.

The above example has interestingly demonstrated that previous techniques may not be effective in a common case, while our approach has the potential to address it. In the next section, we will elaborate on our model.

III. OUR APPROACH

In this section, we introduce the problem settings, give definitions, and elaborate on our model PAFL.

A. Definitions

We use the definition of "coverage vector" to formally describe the concept of "distinct path" used in Section II.

[Definition I] *An **original coverage vector** $ocv_i = \langle b_1, b_2, ..., b_n\rangle$ ($b_j \in \{0, 1\}$ for $j = 1, 2, ..., n$) of program execution $P(t_i)$ is a tuple. We use $ocv_i(b_j)$* to retrieve the *j*-th 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 $P(t_i)$, we also say $P(t_i)$ covers ocv_i.*

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

0, 0, 1, 0, 1, 0, 1⟩. Apparently, an original coverage vector can be covered by many different executions (even by both some passed executions and some failed executions). So let us move to Definition II.

[Definition II] *The **distinct coverage vector set** $CV = \{cv_1, cv_2, ..., cv_p\}$ is the distinct set (with no repeating elements) of all original coverage vectors ocv_i with respect to the program execution $P(t_i)$ of each test case t_i. Each element $cv_i \in CV$ is called a **coverage vector**. Similarly, we use $cv_i(b_j)$ to retrieve the j-th 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 ($1 \leq i < j \leq p$). In Figure 1, there are two coverage vectors, namely, cv_1 = ⟨1, 0, 0, 0, 0, 1, 0, 1, 0, 1⟩ and cv_2 = ⟨1, 1, 0, 1, 0, 0, 0, 0, 0, 1⟩.

B. Failing Rate of Coverage Vector

Since a coverage vector may be covered by both passed executions and failed executions, we are also interested in the ratio of failed executions that covers the coverage vector (to all executions that covers the coverage vector). We use the term *failing rate of a coverage vector* to denote such a ratio, which is calculated using equation (1).

$$\theta(cv_i) = \frac{failed(cv_i)}{failed(cv_i) + passed(cv_i)} \quad (1)$$

In equation (1), *failed* (cv_i) and *passed* (cv_i) respectively refer to the number of failed and passed executions that cover cv_i. They are calculated using equation (2) and (3).

$$failed(cv_i) = \left|\{v_j | P(v_j) \text{ covers } cv_i\}\right| \quad (2)$$

$$passed(cv_i) = \left|\{u_j | P(u_j) \text{ covers } cv_i\}\right| \quad (3)$$

Here, we adopt the formula of SBI in equation (1) because it gives a best estimation to the probability of an exercised program entity causing a failure [14].

For a coverage vector cv_i with $\theta(cv_i)$ greater than zero, it indicates that cv_i is covered by at least one failed execution, and it is also denoted as a *failed coverage vector*. For a coverage vector cv_i with $\theta(cv_i)$ equals to zero, it indicates that cv_i is covered by no failed execution, and it is also denoted as a *passed coverage vector*.

According to equation (1), the failing rates of cv_1 and cv_2 are $\theta(cv_1)$ = 0.25 and $\theta(cv_2)$ = 0.50, respectively. Both of them are failed coverage vectors.

C. Suspiciousness Scores of Blocks

After we have identified all the coverage vectors, we also need to calculate suspiciousness scores for basic blocks. Inspired by previous study [6], we employ the Jaccard similarity coefficient to evaluate the suspiciousness scores for basic blocks, by contrasting the execution spectra of basic blocks on the coverage vector level. In this paper, we use the term $susp(b_i)$ to denote such suspiciousness score of basic block b_i, which is calculated using equation (4).

$$susp(b_i) = \frac{\left|\{cv_j | \theta(cv_j) > 0 \wedge cv_j(b_i) = 1\}\right|}{\left|\{cv_j | \theta(cv_j) > 0 \vee cv_j(b_i) = 1\}\right|} \quad (4)$$

The numerator represents the number of failed coverage vectors that cover b_i, the denominator represents the number of coverage vectors that are either failed coverage vectors or cover b_i. Here, we adopt the similarity coefficient Jaccard because it has mature mathematical basis. Further, it has been used in previous techniques and empirically shown effective in locating faults in programs [1].

Equation (4) estimates 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 fault. According to equation (4), we can recall the motivating example in Section II and revisit the suspiciousness scores calculated in Section II. The suspiciousness scores for b_1 is calculated as $\frac{2}{2+2} = 0.50$.

D. Tie breaking

After all the blocks are sorted according to their suspiciousness of relating to fault and form a list, programming may search along the generated list for the fault. Particularly, when some basic blocks have identical suspiciousness scores, we use equation (5) to break tie.

$$conf(b_i) = \frac{\sum_{cv_j(b_i)=1}[\theta(cv_j)]}{\left|\{cv_j | \theta(cv_j) > 0, cv_j(b_i) = 1\}\right|} \quad (5)$$

Equation (5) calculates the average failing rate of the coverage vectors that exercising basic block b_i. The rational is that for two basic blocks having identical probability of causing failure, we deem the one whose appearance in a path has higher chance to reveal a failure as more related to faults.

For example, in Figure 1, basic blocks b_1 and b_{10} form a tie. We calculate that $conf(b_1) = conf(b_{10}) = \frac{0.25+0.5}{2} = 0.375$ so that the tie still cannot be break and thus b_1 and b_{10} are evaluated as a whole. Finally, we need to examine 33% of all code to locate the fault.

IV. EVALUATION

A. Experiments Setup

In this paper, we use the 7 Siemens programs and 3 UNIX utilities to evaluate our technique. Each of them has several faulty versions (downloaded from the SIR repository [3]). They have been used in previous studies [9][13][14]. Table 1 shows the statistics of the subject programs used in the experiments.

In our experiment, we select techniques Tarantula [7], Jaccard [1], SBI [13], SAFL [4], and ICST10 [10] to compare with. Tarantula is an old technique and has a lot of variants [7][13]. Jaccard is evaluated very effective in

Table 1. Statistics of subjects

Subjects	# of faulty versions	# of test cases	Description
print_tokens	7	4130	lexical analyzer
print_tokens2	10	4115	lexical analyzer
replace	32	5542	pattern replacement
schedule	9	2650	priority scheduler
schedule2	10	2650	priority scheduler
tcas	40	1578	altitude separation
tot_info	23	1054	information measure
flex	56	567	lexical parser
grep	21	809	text processor
gzip	18	213	compressor
in total	226		

previous studies [10][14]. SBI is the statement-level version of CBI [9], while the latter is a classic predicate-level technique. SAFL and ICST10 investigate execution similarity to reduce the noise from coincidental correctness and relates to them.

B. *Effectiveness on Subject Programs*

To know the overall effectiveness of the studied techniques, we take the average of the 10 programs to show in Figure 2. In Figure 2, the x-coordinates mean the percentage of code examined in each faulty version; the y-coordinates show the percentage of faulty versions, in which, faults can be located within the code examining effort specified by the x-coordinates.

From Figure 2, we observe that at most checkpoints (except the 50% checkpoint), PAFL is more effective than, or at least comparable to, the other techniques. For example, on average, by examining up to 5% of all the code in faulty versions, PAFL can locate faults in 34% of all faulty versions, Jaccard can locate 33%, Tarantula can locate 23%, SBI can locate 22%, SAFL can locate 6%, and ICST10 can locate 26%. It shows that PAFL has an overall better effectiveness than the other techniques studied.

Table 2 shows the mean effectiveness of these techniques on each program. Limited by the space, we cannot show results of the minimum, maximum, and standard deviation measurements. This table shows that for these programs, PAFL is often, but not always, the best among the four techniques.

V. CONCLUSION

In this paper, we demonstrate that frequently occurred execution similarity may affect the effectiveness of existing fault localization techniques and propose PAFL to alleviate the impact of coincidental correctness. We use the concept of coverage vector to count distinct execution paths, and calculate failing rate for each coverage vector, thus refine the executions spectra to reduce noise. The empirical study shows that our technique outperforms five previous techniques on Siemens and UNIX programs. The experiment also shows that our technique is particularly effectiveness to alleviate the impact of execution similarity and coincidental correctness. The future work is to investigate the impact of test case selection to PAFL and adapt PAFL to locate faults in multi-fault programs.

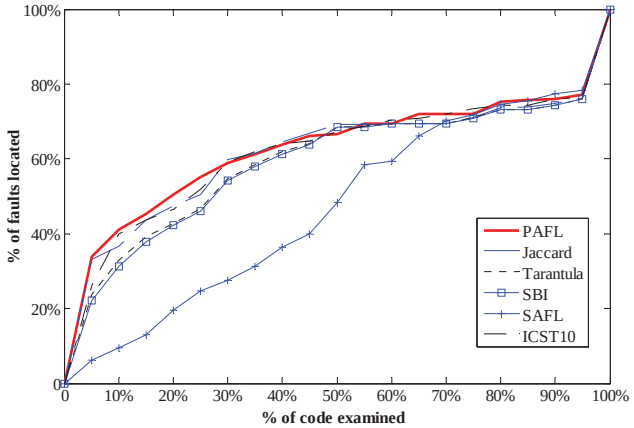

Figure 2. Overall effectiveness

Table 2. Mean effectiveness on individual programs

Subjects	PAFL	Jaccard	Tarantula	SBI	SAFL	ICST10
print_tokens	**72%**	74%	77%	77%	84%	74%
print_tokens2	**22%**	24%	25%	25%	55%	24%
replace	24%	**21%**	24%	24%	37%	**21%**
schedule	**23%**	24%	25%	25%	53%	24%
schedule2	85%	85%	85%	85%	**82%**	84%
tcas	54%	56%	58%	58%	66%	**51%**
tot_info	**37%**	43%	47%	47%	64%	43%
flex	**27%**	**27%**	30%	32%	45%	30%
grep	**21%**	**21%**	23%	26%	34%	23%
gzip	**12%**	**12%**	14%	15%	18%	14%

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (nos. 60970114, 61003027, 61073006) and the Scholarship Award for Excellent Doctoral Student granted by Chinese Ministry of Education.

REFERENCES

[1] R. Abreu, P. Zoeteweij, and A.J.C.van Gemund. On the accuracy of spectrum-based fault localization. I*n Proc. of Testing: Academic and Industrial Conference, Practice and Research Techniques.*

[2] B Baudry, F Fleurey, and Y. Traon. Improving Test Suites for Efficient Fault Localization. In *Proc. of ICSE'06*.

[3] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Software Engineering*, 2005.

[4] D. Hao, L. Zhang, Y. Pan, H. Mei, and J. Sun. On similarity-awareness in testing-basedfault localization. *JASE*, 2008.

[5] R. M. Hierons. Avoiding coincidental correctness in boundary value analysis. *TOSEM.*, 2006.

[6] P. Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. Bulletin del la Socit Vaudoise des Sciences Naturelles 37.

[7] J. A. Jones and M. J. Harrold. Visualization of test informationto assistfault localization. In *Proc. of ICSE'02*.

[8] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automaticfault-localization technique. In *Proc. of ASE'05*.

[9] B. Liblit, A. Aiken, A. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proc. of PLDI'03*.

[10] W. Masri, and R. Assi. Cleansing Test Suites from Coincidental Correctness to Enhance Fault-Localization. In *Proc. of ICST'10*.

[11] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold. Lightweightfault localization using multiple coverage types. In *Proc. of ICSE'09*.

[12] 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 *Proc. of ICSE'09*.

[13] Y. Yu, J. A. Jones, and M. J. Harrold. An empirical study of the effects of test-suite reduction on fault localization. In *Proc. of ICSE'08*.

[14] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In *Proc. of FSE/ESEC'09*.