Accepted by IEEE Transactions on Reliability, https://doi.org/10.1109/TR.2020.2982975

Improving Fault-localization Accuracy by Referencing Debugging History to Alleviate Structure Bias in Code Suspiciousness

Long Zhang, *Student Member, IEEE,* Zijie Li, *Student Member, IEEE,* Yang Feng, *Member, IEEE,* Zhenyu Zhang, *Member, IEEE,* W. K. Chan, *Member, IEEE,* Jian Zhang, *Senior Member, IEEE,* and Yuming Zhou, *Member, IEEE*

Abstract—Spectrum-Based Fault Localization (SBFL) techniques can automatically localize software faults. They employ the program spectrum, such as code coverage profile with test verdicts, to rank the program entities based on their code suspiciousness. In the past decades, researchers have proposed many approaches to optimize these techniques; however, the program structure, which can influence their performance, is not taken into consideration in developing and improving these techniques. In this work, we identify and analyze the effect of the program structure on the application of SBFL techniques. We observe that some specific program structures may introduce *structure bias* to code suspiciousness and negatively influence the output of SBFL techniques. To mitigate these effects and improve the performance of fault localization, we propose Delta4Ts, a structure-aware technique. Delta4Ts references debugging history to alleviate the impact of structure bias in the calculation of code suspiciousness. It reasons from the observable suspicious value towards the desired suspicious value and the impact of structure bias. To evaluate Delta4Ts under practical constraints, we conduct a controlled experiment using nine widely-studied SBFL formulae on 12 C programs and 6 Java programs. The experiment results show that Delta4Ts can significantly improve the accuracy of the studied SBFL formulae by an average of 34.8% on 12 C programs and 30.6% on 6 Java programs, and improve more on subject programs associated with more history versions or having larger code sizes.

Index Terms—Software Testing, Program Debugging, Spectrum-based Fault Localization.

1 Introduction

With the tremendous increase in both complexity and size of modern software, debugging has become a challenging yet inevitable task [58], [92]. The high cost of manual debugging has naturally motivated the development of full- or semi-automated techniques. Among these techniques, Spectrum-Based Fault Localization (**SBFL** in short) techniques have been under active research, and have demonstrated the effectiveness of improving the efficiency of localizing faults. SBFL techniques, such as Tarantula [38], Jaccard [1], and Ochiai [63], are designed to automatically guide developers' attention to specific parts of a target program. They leverage the coverage spectrum collected in the testing process to estimate the suspiciousness of source code statements, aiming to localize the statements that are most suspicious to be erroneous and associated with faults.

Typically, an SBFL technique makes use of two kinds of information collected in the testing process, i.e., test verdicts and program coverage spectra, to estimate the suspiciousness of each program entity. The test verdict of a test case represents whether a failure is detected from the corresponding test output or not. A program coverage spectrum is a collection of data that provides a specific view of the dynamic behavior of the program during execution, which records the execution profiles of program entities for a specific test suite, where program entities can be statements, branches, basic blocks, and so on [34], [74].

Taking Tarantula [39] as an example, its SBFL formula is shown in Equation 1.

$$\mathcal{S}_{\text{Tarantula}}(s) = \frac{\frac{\mathrm{a_{ef}}}{\mathrm{a_{ef}} + \mathrm{a_{nf}}}}{\frac{\mathrm{a_{ef}}}{\mathrm{a_{ef}} + \mathrm{a_{nf}}} + \frac{\mathrm{a_{ep}}}{\mathrm{a_{ep}} + \mathrm{a_{np}}}} \quad (1)$$

In Equation 1, given a program entity s, $\mathrm{a_{ef}}$ and $\mathrm{a_{nf}}$ denote the numbers of failed tests that execute and do not execute s in the execution of the test suite, respectively. Similarly, $\mathrm{a_{ep}}$ and $\mathrm{a_{np}}$ respectively denote the numbers of passed tests that execute and do not execute s. The basic idea of Tarantula is as follows: Each test execution contributes certain code coverage statistics on each program entity

- *Long Zhang is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China. He is also with the University of Chinese Academy of Sciences, Beijing, China.*
- *Zijie Li is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China. He is also with the University of Chinese Academy of Sciences, Beijing, China.*
- *Yang Feng is with the Department of Computer Science and Technology, Nanjing University, Nanjing, China.*
- *Zhenyu Zhang is the corresponding author. He is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China. He was also with the Institute for Software Research, University of California, Irvine, Irvine, CA USA, when he was preparing the first version of the paper. E-mail: zhangzy@ios.ac.cn*
- *W. K. Chan is with the Department of Computer Science, City University of Hong Kong, Hong Kong.*
- *Jian Zhang is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China. He is also with the University of Chinese Academy of Sciences, Beijing, China.*
- *Yuming Zhou is with the Department of Computer Science and Technology, Nanjing University, Nanjing, China.*

Manuscript received March 20, 2020; revised March 20, 2020.

in the program; Tarantula firstly measures to what extent the code coverage achieved by both passed test runs and failed test runs correlates to individual program entities through its SBFL formula. Then, it computes a correlation value (aka, a *suspiciousness score* [38], [39]) on a program entity as the indicator of that program entity correlating to the failures. Finally, it ranks all these program entities in descending order of their suspiciousness scores, with or without tie-breaking mechanisms. The resultant ranked list can save developers many efforts by guiding them to focus on suspicious parts.

In the past decades, SBFL techniques are under active research and tremendous advancement has been made. Researchers have proposed many new SBFL approaches [1], [38], [46], [49], [113], [119] and extensively conducted empirical studies to investigate their performance [28], [58], [104], [112]. Meanwhile, a number of methods are designed for addressing the practical issues inherent to the execution profiles, such as coincidental correctness [58], [75], background noise [46], [104] and class imbalance problem [117], Also, several groups theoretically investigate the correlation between different SBFL formulae [60], [62], [103], [108]. Their research moves a step forward in revealing the reasons on why the accuracy of one SBFL formula is theoretically related to the accuracy of another SBFL formula.

However, only little research investigates the effect of program structure on the performance of SBFL techniques. Jones et al. [38] have shown that faults lay in statements of those code regions frequently executed by both passed and failed test cases cannot be effectively located by SBFL formulae. Our prior work [47] has further pointed out that an SBFL formula has a tendency to rank a program entity appearing in some structured code fragments higher or lower than some other structured code fragments. For instance, typically, a statement in a *catch block* in an exception handler of a Java program is more likely to be executed in failed program executions than in passed ones. Moreover, the suspiciousness of multiple program entities assessed by an SBFL formula may be subject to different dependency relationships imposed by the program structure.

We notice that the program structure inherently influences the SBFL formula and may lead it to assign an inflated (i.e., higher than the actual) suspiciousness score or a deflated one (i.e., lower than the actual) to a program entity. We refer the effects from specific program structures on suspiciousness score of a program entity as **structure bias**. Further, we notice that the suspiciousness score of each program entity can be assessed through a postmortem analysis, and this analysis inspires us to develop this work.

In this paper, we generalize the preliminary work [47] to address the effect of structure bias in multi-fault programs. We formulate the problem of structure bias and present a theoretical model to analyze its effect in multi-fault programs. Our model is built upon the empirical observation and signal theory, and it models the two fundamental components of the actual suspiciousness score, i.e., the faultiness of program entities and the effect brought by structure bias, as the *desired signal* and *false signal*, respectively. Further, we implement our model, namely **Delta4Ts**, to mitigate the negative effects of structure bias. Note that our model is designed for software development environments in which program version histories and debugging records are available.

We experiment Delta4Ts on nine widely-studied SBFL formulae and 18 subjects under practical constraints. The experimental results show that Delta4Ts can significantly improve the accuracy of these SBFL formulae by 34.8%, and improve considerably more on faulty programs associated with more versions of structure variations, or having larger code sizes.

In comparison with the conference version, in this paper, we make the following contributions:

- This work systematically investigates the effect of program structures on the performance of SBFL techniques. We model *structure bias* with signal theory and comprehensively analyze its effects. We formalize the model and provide proof and derivation to lay theoretical foundation of our model.
- We present a solution, namely Delta4Ts, to mitigate the negative effect of structure bias and improve the performance of SBFL techniques. We also analyze the factors that can influence our model.
- Based on 12 benchmark C programs and 6 real Java programs, we extensively conduct empirical study on nine classic SBFL formulae to discuss the effect of structure bias, and further validate our solution.

The rest of this paper is organized as follows. Section 2 uses a series of examples to demonstrate the problem and motivate our work. Section 3 presents our model, which is evaluated in Section 4. Section 5 reviews related work. Section 6 concludes this paper and outlines the directions for future work.

2 MOTIVATION

2.1 Motivating Example

In this section, we first present a program to exemplify common program structures. Based on it, we discuss the motivation of our work by demonstrating the problem of how the accuracy of SBFL techniques is affected by program structures. And then, we analyze the cause of this problem and introduce our solution.

We provide an example to show how the program structure influence the performance of SBFL. Fig. 1 lists out the same code blocks from three versions, denoted as $\mathcal{V}^1$, $\mathcal{V}^2$, and $\mathcal{V}^3$, of the program tcas, which can be downloaded from the SIR benchmark [17]. Based on the information provided by the SIR, $\mathcal{V}^3$ is newer than $\mathcal{V}^1$ and $\mathcal{V}^2$. Thus, we can safely assume that the current working version is $\mathcal{V}^3$ and the historic information of $\mathcal{V}^1$ and $\mathcal{V}^2$ is available.

Four basic blocks $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$, and $\mathcal{B}_4$ of these three versions are shown in the code excerpt. More specifically, $\mathcal{B}_1$ is a basic block enclosed in an if branch. It assigns an initial value to an array variable whenever the branch is taken to execute. $\mathcal{B}_2$ is a basic block enclosed in a while loop, which iterates over an array and updates its elements. $\mathcal{B}_3$ is a basic block enclosed in an if branch belonging to a while loop. It is used to update the program states subject to certain conditions. $\mathcal{B}_4$ is a basic block enclosed in a try-catch exception handler, which outputs exception messages. These four basic blocks are listed in the order of their appearance in the program.

	History versions						Current version		
	Version $\mathcal{V}^1$ (10 of 20 test cases failed)	a_{ef}	a_{ep}	Version $\mathcal{V}^2$ (7 of 20 test cases failed)	a_{ef}	a_{ep}	Version $\mathcal{V}^3$ (5 of 20 test cases failed)	a_{ef}	a_{ep}

$\mathcal{B}_1$	`if(array) {`
	`//array[0]=1;`
	`array[1]=1; }`
		10	5	`if(array) {`
				`··· }`
					7	11	`if(array) {`
							`··· }`
								5	9

$\mathcal{B}_2$	`while(...) {`
	`··· }`
		5	9	`while(...) {`
				`//a[i]=b;`
				`a[i]=c; }`
					7	5	`while(...) {`
							`··· }`
								5	10

$\mathcal{B}_3$	`while(...) {`
	`if(...) {`
	`··· } }`
		6	6	`while(...) {`
				`if(...) {`
				`··· } }`
					5	6	`while(...) {`
							`if(...) {`
							`//z=x*10;`
							`z=z*10; } }`
								5	9

$\mathcal{B}_4$	`catch() {` `println(e); }`	7	2	`catch() {` `println(e); }`	3	1	`catch() {` `println(e); }`	4	2

Fig. 1: Three Faulty Versions of an Example Program

Note that, because the functionality of $\mathcal{B}_1$ is to initialize the *array* that is required for the execution of the $\mathcal{B}_2$ and $\mathcal{B}_3$, $\mathcal{B}_1$ has a higher or equal chance to be executed in comparison with $\mathcal{B}_2$ and $\mathcal{B}_3$. And $\mathcal{B}_4$ is used to handle exceptions which can only be covered when an exception is thrown. Thus, it has a high probability of being covered by failed test cases.

SIR documents the bug information of these three versions, respectively: the version $\mathcal{V}^1$ contains a fault "`array[1]=1`" (the correct code should be "`array[0]=1`") in the block $\mathcal{B}_1$, which may cause a memory location left uninitialized; The version $\mathcal{V}^2$ contains a fault "`a[i]=c`" (the correct code should be "`a[i]=b`") in the block $\mathcal{B}_2$, which assigns incorrect values to array elements; And the version $\mathcal{V}^3$ contains a fault "`z=z*10`" (the correct code should be "`z=x*10`") in the block $\mathcal{B}_3$, which calculates a value by mistake. We download 20 test cases from SIR, and run all of them on each of these versions. During the procedure, we collect the execution traces and verdicts for each of them. On version $\mathcal{V}^1$, ten of them failed and the rest passed; On version $\mathcal{V}^2$ seven of them failed and the rest passed; And on version $\mathcal{V}^3$, five of them failed and the rest passed.

In this figure, we employ two variables, a_{ef} and a_{ep}, to denote the the number of failed test cases and the number of passed test cases for a basic executed block. We can observe that, for code blocks, the probability of being executed is attributed to the program structure: while $\mathcal{B}_1$ has been executed by most test cases (at least 14/20 test cases across these three versions) and more often than $\mathcal{B}_2$ and $\mathcal{B}_3$ in $\mathcal{V}^1$ and $\mathcal{V}^2$, the exception handler block, $\mathcal{B}_4$, is executed more often by failed test cases than by passed test cases, i.e., 7 failed cases to 2 passed cases in $\mathcal{V}^1$, 3 failed cases to 1 passed case in $\mathcal{V}^2$, and 4 failed cases to 2 passed cases in $\mathcal{V}^3$.

While the program structure has impacts on the performance of SBFL techniques, we employ two classic SBFL techniques, i.e., Tarantula and Wong1, to demonstrate it. Recall that we have shown the SBFL formula of Tarantula in Equation 1. Wong1 is an SBFL formula proposed in [100] and is known to be a maximal formula [89], [103].

The equation for computing suspiciousness score of Wong1 is relatively simple, which is shown below.

$$\mathcal{S}_{\mathsf{Wong1}} = a_{ef} \tag{2}$$

Based on coverage information shown in Figure 1, we present the output of Tarantula and Wong1, i.e., ranking result, as well as suspicious score, in the conventional solution column of Table 1.[1] Here, S^k represents the set of suspicious values for each basic block of version $\mathcal{V}^k$ in Table 1. We first analyze the computation result of Wong1 that is shown in the top-left part of this table. In $\mathcal{V}^1$, Wong1 assigns the highest suspiciousness score for $\mathcal{B}_1$. Accordingly $\mathcal{B}_1$ ranks 1st among the four basic blocks, and the fault in $\mathcal{B}_1$ is easily located since $\mathcal{B}_1$ will be checked first. However, because $\mathcal{B}_1$ is a block that is frequently executed and accordingly has a high a_{ef} value, we cannot identify which one is the dominant factor for resulting in the high effectiveness of Wong1. For version $\mathcal{V}^2$, their spectra show the suspiciousness scores for the four basic blocks are 7, 7, 5, and 3. This means $\mathcal{B}_1$ and $\mathcal{B}_2$ share the same rank in $\mathcal{V}^2$. Thus, $\mathcal{B}_1$ may mislead the manual inspection process. $\mathcal{V}^3$ shows similar results. All the failed test cases executing $\mathcal{B}_3$ passed through $\mathcal{B}_1$ and $\mathcal{B}_2$, and happened to execute them in this example. Finally, in $\mathcal{V}^3$, $\mathcal{B}_1$, $\mathcal{B}_2$ and $\mathcal{B}_3$ have identical suspiciousness scores and share the same rank, which makes Wong1 ineffective to locate the fault in $\mathcal{B}_3$. This analysis shows code blocks that are covered by most test cases can be inflated in Wong1 and mislead the fault localization process.

The lower left part of Table 1 shows the calculation of the suspiciousness scores for the four basic blocks in the three versions using the Tarantula formula. In $\mathcal{V}^1$, since a_{ef} and a_{ep} for $\mathcal{B}_1$ are 10 and 5 in $\mathcal{V}^1$ (as shown in Fig. 1),

1. If a block is assigned value v by an SBFL similarity coefficient, it denotes the position of the block is v-th in ranked list. In Table 1, "rank=i" denotes the position of the faulty block is v-th in the ranked list.

TABLE 1: Conventional Solution v.s. Delta4Ts

		Conventional Solution			Our Solution Delta4Ts					
					$\mathcal{V}^3$ v.s. average (to locate $\mathcal{B}_3$)		*$\mathcal{V}^3$ v.s. individual (to locate $\mathcal{B}_3$)*		*$\mathcal{V}^1$, $\mathcal{V}^2$ v.s. average (to locate $\mathcal{B}_1$ and $\mathcal{B}_2$, respectively)*	
		$\mathcal{S}^1$	$\mathcal{S}^2$	$\mathcal{S}^3$	$\frac{\mathcal{S}^1+\mathcal{S}^2}{2}$	$\mathcal{S}^3 - \frac{\mathcal{S}^1+\mathcal{S}^2}{2}$	$\mathcal{S}^3 - \mathcal{S}^1$	$\mathcal{S}^3 - \mathcal{S}^2$	$\mathcal{S}^1 - \frac{\mathcal{S}^2+\mathcal{S}^3}{2}$	$\mathcal{S}^2 - \frac{\mathcal{S}^1+\mathcal{S}^3}{2}$
by Wong1	$\mathcal{B}_1$	10	7	5	8.50	−3.50	−5.00	−2.00	4.00	−0.50
	$\mathcal{B}_2$	5	7	5	6.00	−1.00	0.00	−2.00	−1.00	2.00
	$\mathcal{B}_3$	6	5	5	5.50	−0.50	−1.00	0.00	1.00	−0.50
	$\mathcal{B}_4$	7	3	4	5.00	−1.00	−3.00	1.00	3.50	−2.50
						rank = 1	*rank = 2*	*rank = 2*	*rank = 1*	*rank = 1*
by Tarantula	$\mathcal{B}_1$	0.67	0.54	0.63	0.60	0.03	−0.04	0.08	0.08	−0.10
	$\mathcal{B}_2$	0.36	0.72	0.60	0.54	0.06	0.24	−0.12	−0.30	0.24
	$\mathcal{B}_3$	0.50	0.61	0.63	0.55	0.07	0.13	0.02	−0.12	0.04
	$\mathcal{B}_4$	0.78	0.85	0.86	0.81	0.04	0.08	0.01	−0.07	0.03
						rank = 1	*rank = 2*	*rank = 2*	*rank = 1*	*rank = 1*

respectively, $\mathcal{S}_{\mathsf{Tarantula}}$ for $\mathcal{B}_1$ is calculated as $\frac{\frac{10}{10}}{\frac{10}{10}+\frac{5}{10}} \approx 0.67$ in $\mathcal{V}^1$. The suspiciousness scores for the other basic blocks in $\mathcal{V}^1$ are similarly calculated and shown in the table (i.e., 0.36 for $\mathcal{B}_2$, 0.50 for $\mathcal{B}_3$, and 0.78 for $\mathcal{B}_4$). Although $\mathcal{B}_1$ is assigned a higher suspiciousness score than $\mathcal{B}_2$ and $\mathcal{B}_3$, $\mathcal{B}_1$'s suspiciousness score is still lower than the suspiciousness score of $\mathcal{B}_4$ (i.e., 0.78 in this example). The reason is that $\mathcal{B}_4$ is used to output exception messages in this example and hence is executed very often by failed test cases but rarely by passed test cases. In consequence, $\mathcal{B}_4$ is ranked ahead of $\mathcal{B}_1$. When we move to the versions $\mathcal{V}^2$ and $\mathcal{V}^3$, the situations are similar in that (1) $\mathcal{B}_2$ is assigned a higher suspiciousness score than $\mathcal{B}_1$ and $\mathcal{B}_3$ in $\mathcal{V}^2$, (2) $\mathcal{B}_3$ is assigned a higher suspiciousness score than $\mathcal{B}_2$ in $\mathcal{V}^3$, and (3) $\mathcal{B}_4$ is always assigned the highest suspiciousness score in $\mathcal{V}^2$ and $\mathcal{V}^3$. As a result, the basic blocks $\mathcal{B}_2$ in $\mathcal{V}^2$ and $\mathcal{B}_3$ in $\mathcal{V}^3$ are less likely to be ranked high by Tarantula. In summary, since $\mathcal{B}_4$ is used to report failure in this example, it is often associated with failed test cases and assigned a high suspiciousness score by Tarantula.

While the analysis of the output of these two classic SBFL techniques exemplifies that common program structures may have impacts on the accuracies of SBFL formulae when assessing different program entities, the extents of the corresponding impacts can be different and many program entities are affected by such impacts. Is there any method to alleviate the impact of such common program structures on the accuracies of SBFL formulae? To the best of our knowledge, no existing SBFL research (except the preliminary version of the present paper [47]) focuses on this issue.

In next section, we introduce our solution.

2.2 How Delta4Ts Addresses Such Impacts

In this section, we give an idea to inspire Delta4Ts, which uses history versions of the program under debugging as the references to adjust suspiciousness scores in the current version, aiming to alleviate the impacts of program structures.

We consider the calculated suspiciousness score in the current version as a superposition of the *desired* suspiciousness component (that reflects how much a basic block is related to fault) and *false* suspiciousness component (that comes from the impacts of program structures). We capture the latter by averaging the suspiciousness scores of each basic block across all history versions, and remove it from the calculated suspiciousness scores of each basic block in the current version to restore the former. In the right part of Table 1, we present the computation results of our solution. Note that, in this part, to ease explanation, we highlight the ranking of the faulty module below the score.

2.2.1 Pairing with Wong1

The top right part of Table 1 demonstrates our idea using the Wong1 formula. We first calculate the average suspiciousness score for each basic block in the history versions. Let us take the basic block $\mathcal{B}_1$ as an example. The suspiciousness scores of $\mathcal{B}_1$ in $\mathcal{V}^1$ and $\mathcal{V}^2$ are 10 and 7, respectively, and their average is calculated as $\frac{10+7}{2} = 8.50$. Similarly, the column "$\frac{\mathcal{S}^1+\mathcal{S}^2}{2}$" also lists out the average suspiciousness scores of $\mathcal{B}_2$, $\mathcal{B}_3$, and $\mathcal{B}_4$ in the history versions, which are 6.00, 5.50, and 5.00, respectively. We regard such an average suspiciousness score as the *usual case* for the suspiciousness of a basic block. By looking at such usual case scores, we find that (1) the suspiciousness score of $\mathcal{B}_1$ is usually higher than those of $\mathcal{B}_2$, $\mathcal{B}_3$, and $\mathcal{B}_4$, and (2) the suspiciousness score of $\mathcal{B}_2$, $\mathcal{B}_3$, and $\mathcal{B}_4$ are usually comparable to one another.

Using such usual case scores as the references, we contrast the suspiciousness scores calculated in the current version $\mathcal{V}^3$ with them to restore the desired suspiciousness scores. In particular, we subtract them from the corresponding calculated suspiciousness scores in $\mathcal{V}^3$ to compute the differences, which are shown in the column "$\mathcal{S}^3 - \frac{\mathcal{S}^1+\mathcal{S}^2}{2}$". Note that in Table 1, "rank=i" denotes the fault is the i-th suspicious element in the corresponding ranked list. For example, in column "$\mathcal{V}_3$ v.s. average" and row "by Wong1" of Table 1, "rank=1" denotes that the fault "$\mathcal{B}_3$" is first

element in the ranked list by using $\mathcal{S}^3 - \frac{\mathcal{S}^1+\mathcal{S}^2}{2}$ method to capture the impact of structure bias. By checking the adjusted suspiciousness scores (i.e., -3.50 for $\mathcal{B}_1$, -1.00 for $\mathcal{B}_2$, -0.50 for $\mathcal{B}_3$, and -1.00 for $\mathcal{B}_4$), the basic block $\mathcal{B}_3$ is found having the highest suspiciousness score and accordingly assigned as rank 1. As a result, $\mathcal{B}_3$ is ranked ahead of $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_4$ in $\mathcal{V}^3$.

We next move to check the SBFL formula Tarantula.

2.2.2 Pairing with Tarantula

We repeat the above process and show the calculation in the lower right part of Table 1. The calculated average suspiciousness scores for the four basic blocks $\mathcal{B}_1$-$\mathcal{B}_4$ are 0.60, 0.54, 0.55, and 0.81, respectively. We find that (1) the suspiciousness score of $\mathcal{B}_4$ is usually higher than those of $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$, and (2) the suspiciousness score of $\mathcal{B}_1$, $\mathcal{B}_2$, and $\mathcal{B}_3$ are usually comparable to one another. By subtracting the corresponding usual case scores from the calculated suspiciousness scores in $\mathcal{V}^3$, we get the adjusted suspiciousness scores (i.e., 0.02 for $\mathcal{B}_1$, 0.06 for $\mathcal{B}_2$, 0.07 for $\mathcal{B}_3$, and 0.04 for $\mathcal{B}_4$), and the basic block $\mathcal{B}_3$ is found having the highest suspiciousness and accordingly assigned as rank 1. As a result, the fault in $\mathcal{B}_3$ is easily located.

We further find that (1) by comparing the suspiciousness scores of basic blocks in $\mathcal{V}^3$ with those suspiciousness scores from an individual history version (either $\mathcal{V}^1$ or $\mathcal{V}^2$), the method does not work, and (2) by referencing other two versions as history versions to locate the fault in the remaining version, the method still works. The related calculation are also shown in the table. Using $\mathcal{V}^1$ as history version to locate the fault in $\mathcal{V}^3$, the adjusted suspiciousness scores by Wong1 for the four basic blocks are -5.00, 0.00, -1.00, and -3.00, respectively, and the faulty basic block $\mathcal{B}_3$ is only assigned rank 2. Using $\mathcal{V}^2$ as the entire history version to locate the fault in $\mathcal{V}^3$, the adjusted suspiciousness scores by Wong1 for the four basic blocks are -2.00, -2.00, 0.00, and 1.00, respectively, and the faulty basic block $\mathcal{B}_3$ is only assigned as rank 2. Similarly, using an individual version (either $\mathcal{V}^1$ or $\mathcal{V}^2$) as the entire history to locate the fault in $\mathcal{V}^3$, Tarantula also assigns the rank 2 to $\mathcal{B}_3$.

On the contrary, we also conduct two gedankenexperiments, i.e., to use both $\mathcal{V}^2$ and $\mathcal{V}^3$ as the entire history to locate the fault in $\mathcal{V}^1$, and to use both $\mathcal{V}^1$ and $\mathcal{V}^3$ as the entire history to locate the fault in $\mathcal{V}^2$. In both experiments, Wong1 and Tarantula can assign rank 1 to the faulty basic block (i.e., $\mathcal{B}_1$ in $\mathcal{V}^1$ and $\mathcal{B}_2$ in $\mathcal{V}^2$).

Based on the motivating example, we verify whether this approach can improve the effectiveness of other formulae. The motivating example[2] reported that particular program structure entities might receive inflated suspiciousness scores, affecting the accuracies of SBFL formulae. Further, for different SBFL formulae, such impacts may also be different. The example also demonstrated the idea of contrasting history versions as reference to restore the desired suspiciousness scores. It thus successfully removed the impacts from the original suspiciousness scores, and improved the accuracies of SBFL formulae (i.e., Wong1 and Tarantula) on the tcas excerpt (C language subject).

2. In Section 4, we will verify whether this approach can improve the effectiveness of other formulae.

In next section, we present the model behind our model — Delta4Ts, which generalizes the motivating example.

3 The Model behind Delta4Ts

In this section, we present the model behind Delta4Ts to capture and remove the impact of program structure on the performance of SBFL techniques. Our model is designed for software development environments in which program version histories and debugging records are available.

3.1 Problem Settings

Let us consider a program having n basic blocks, i.e., $\langle \mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_n \rangle$. Without loss of generality, we suppose that there are m ($m > 1$) faulty program versions in the development history of the program, of which the identified fault(s) in each program version have been located and subsequently fixed.[3] We suppose that the first ranked fault in rank list is the identified fault, and the other faults are not identified. We further denote the corresponding $m+1$ versions of the program as $\langle \mathcal{V}^0, \mathcal{V}^1, \mathcal{V}^2, \ldots, \mathcal{V}^m \rangle$, where $\mathcal{V}^k$ (for $k \in [0, m-1]$) is a history program version, and $\mathcal{V}^m$ is the current program version under debugging.

Suppose that each fault in a program version $\mathcal{V}^k$ has been assessed by an SBFL technique, and we denote the suspiciousness score computed by the technique for $\mathcal{B}_i$ in $\mathcal{V}^k$ as $\mathcal{S}_i^k$, where $k \in [0, m]$ is the index of the program version, $i \in [1, n]$ is the index of basic block. Note that because the suspicious values of different historical versions may range widely, we normalized the suspiciousness values to the range $[0, 1]$ to make different version information weights consistently.

3.2 A Signal-Superposition Point of View

Give the setting mentioned above, we model the suspiciousness $\mathcal{S}_i$ of a basic block $\mathcal{B}_i$ as a superposition of two kinds of signals: $\mathcal{D}_i$ and $\mathcal{T}_i$.

$$\underbrace{\mathcal{S}_i}_{\text{obtained signal}} = \underbrace{\mathcal{D}_i}_{\text{desired signal}} + \underbrace{\mathcal{T}_i}_{\text{false signal}} \tag{3}$$

We present Equation 3 to detail the modeling procedure. In Equation 3, $\mathcal{S}_i$ denotes the suspiciousness of block $\mathcal{B}_i$ computed by an SBFL technique. In our model, $\mathcal{S}_i$ is computed as the sum of $\mathcal{D}_i$ and $\mathcal{T}_i$. $\mathcal{D}_i$ denotes the observable suspiciousness signal beyond structure bias, and $\mathcal{T}_i$ reflects the impact of structure bias. We use the history versions to approximate $\mathcal{T}_i$. Note that no fault has been identified in $\mathcal{B}_i$ in every such history version, which approximates the structure bias incurred by $\mathcal{B}_i$. In signal processing, $\mathcal{T}_i$ is referred to as a *false signal*, and $\mathcal{D}_i$ is the *desired signal*. To assess $\mathcal{B}_i$, we compute $\mathcal{D}_i$ from $\mathcal{S}_i - \mathcal{T}_i$.

Since our model contrasts multiple versions to locate faults, we name it *Delta4Ts* to reflect its mechanism. The Delta4Ts workflow is depicted in Figure 2.

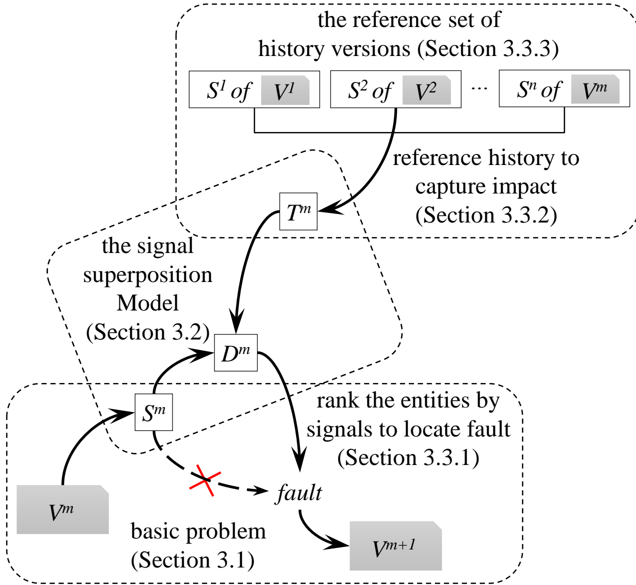


Fig. 2: Workflow of Our Method Delta4Ts

3.3 Our Proposal: Delta4Ts

In the lower part of Figure 2, we show the workflow of the conventional solution, where a program version $\mathcal{V}^m$ is under debugging and the suspiciousness $\mathcal{S}^m$ is generated to prioritize the search region to locate a fault. After a fault is located and fixed, the next version $\mathcal{V}^{m+1}$ is synthesized. [4] The central rectangle area shows the main difference between our solution and the conventional solution. In our solution, we do not rank the suspicious program entities by $\mathcal{S}^m$ (a "×" sign is marked on the dashed arrow in the figure). Rather, we calibrate $\mathcal{S}^m$ to $\mathcal{D}^m$ by removing the $\mathcal{T}^m$ captured from the history versions, and use $\mathcal{D}^m$ to rank program entities. The upper part of Figure 2 illustrates that $\mathcal{D}^m$ is got from selected history versions, which is named a *reference set* in our model. Each step of the cycle is elaborated on in the following subsections.

3.3.1 Modeling the Impact of Structure Bias

Definition 1 (Impact of Structure Bias)**.** With respect to an individual basic block $\mathcal{B}_i$, we parameterize Equation 3 as $\mathcal{S}_i^m = \mathcal{D}_i^m + \mathcal{T}_i^m$ to express that $\mathcal{S}_i^m$ consists of two components: $\mathcal{D}_i^m$ is the extent of the basic block $\mathcal{B}_i$ explaining the observed failures in $\mathcal{V}^m$, and $\mathcal{T}_i^m$ is the magnitude brought by structure bias, which is also named the *impact of structure bias* in this paper.

Our goal is to rank all the basic blocks to reflect the extent of their suspiciousness being related to fault. For this purpose, we compare two basic blocks $\mathcal{B}_i$ and $\mathcal{B}_j$ by their suspiciousness extents $\mathcal{D}_i^m$ and $\mathcal{D}_j^m$. We thus define the following term

$$\overline{\mathcal{D}_{i,j}^m} \stackrel{\text{def}}{=} \mathcal{D}_i^m - \mathcal{D}_j^m. \tag{4}$$

3. Each faulty program version contains exactly one fault.

4. Note that we do not consider whether test cases have been changed in different versions. In our model, we use historical debug evaluation values without distinguishing test case changes. This is because we use history suspiciousness values without distinguishing test cases.

The partial order of any two basic blocks can be determined by referencing the value of $\overline{\mathcal{D}_{i,j}^m}$ with respect to them. We thus avoid figuring out the extent of bias for each basic block directly. A positive $\overline{\mathcal{D}_{i,j}^m}$ indicates that the basic block $\mathcal{B}_i$ has a higher desired signal than the basic block $\mathcal{B}_j$. Moreover, the higher the value, the more suspicious $\mathcal{B}_i$ is than $\mathcal{B}_j$. A negative $\overline{\mathcal{D}_{i,j}^m}$ indicates the basic block $\mathcal{B}_i$ has a lower desired signal than the basic block $\mathcal{B}_j$. The lower the value, the less suspicious $\mathcal{B}_i$ is than $\mathcal{B}_j$. A zero $\overline{\mathcal{D}_{i,j}^m}$ denotes the two basic blocks $\mathcal{B}_i$ and $\mathcal{B}_j$ have identical desired signals. Further, $\mathcal{D}_{i,j}^m > 0$, $\mathcal{D}_{i,j}^m = 0$ and $\mathcal{D}_{i,j}^m < 0$ are expressed as partial order tuples $\langle \mathcal{B}_i, \mathcal{B}_j, > \rangle$, $\langle \mathcal{B}_i, \mathcal{B}_j, = \rangle$, and $\langle \mathcal{B}_i, \mathcal{B}_j, < \rangle$, respectively. For example, $\langle \mathcal{B}_i, \mathcal{B}_j, > \rangle$ denotes that the position of $\mathcal{B}_i$ is higher than that of $\mathcal{B}_j$ in ranked list. Then, we use these partial order tuples to generate a ranked list.

When one SBFL technique assesses the basic blocks of one program version, it usually needs to compare different evaluated values, denoted as $\mathcal{S}$, to generate a partially ordered set. Since our model consists of two signal components ($\mathcal{D}$ and $\mathcal{T}$) and binary relationship tuple set may generate a ring structure, [5] readers may wonder whether it compares different desired values ($\mathcal{D}$) to generate such a partially ordered list. The proofs of the three properties (reflexivity, antisymmetry, and transitivity) of a partially ordered list is shown as follows.

1) $\forall i, j, \overline{\mathcal{D}_{i,j}^m} = 0 \implies \overline{\mathcal{D}_{j,i}^m} = 0.$ (Reflexivity)
   Proof:
   $\because \overline{\mathcal{D}_{i,j}^m} = 0 \implies \mathcal{D}_i^m = \mathcal{D}_j^m, \ \therefore \overline{\mathcal{D}_{j,i}^m} = 0.$ □
2) $\forall i, j, \overline{\mathcal{D}_{i,j}^m} \leq 0 \land \overline{\mathcal{D}_{j,i}^m} \leq 0 \implies \overline{\mathcal{D}_{i,j}^m} = 0 \land \overline{\mathcal{D}_{j,i}^m} = 0.$
   Proof: (Antisymmetry)
   $\because \overline{\mathcal{D}_{i,j}^m} \leq 0 \implies \mathcal{D}_i^m - \mathcal{D}_j^m \leq 0 \implies \mathcal{D}_i^m \leq \mathcal{D}_j^m$ and $\overline{\mathcal{D}_{j,i}^m} \leq 0 \implies \mathcal{D}_j^m - \mathcal{D}_i^m \leq 0 \implies \mathcal{D}_j^m \leq \mathcal{D}_i^m$,
   $\therefore \mathcal{D}_j^m = \mathcal{D}_i^m \implies \overline{\mathcal{D}_{i,j}^m} = 0 \land \overline{\mathcal{D}_{j,i}^m} = 0.$ □
3) $\forall i, j, k, \overline{\mathcal{D}_{i,j}^m} \leq 0 \land \overline{\mathcal{D}_{j,k}^m} \leq 0 \implies \overline{\mathcal{D}_{i,k}^m} \leq 0.$
   Proof: (Transitivity)
   $\because \overline{\mathcal{D}_{i,j}^m} \leq 0 \implies \mathcal{D}_i^m \leq \mathcal{D}_j^m$ and $\overline{\mathcal{D}_{j,k}^m} \leq 0 \implies \mathcal{D}_j^m \leq \mathcal{D}_k^m, \ \therefore \mathcal{D}_i^m \leq \mathcal{D}_k^m \implies \overline{\mathcal{D}_{i,k}^m} \leq 0.$ □

We will present how to compute $\overline{\mathcal{D}_{i,j}^m}$ in the next section.

3.3.2 Capturing the Impact

According to Equation 3, $\overline{\mathcal{D}_{i,j}^m}$ is calculated as

$$\begin{aligned}\overline{\mathcal{D}_{i,j}^m} &= \left(\mathcal{S}_i^m - \mathcal{T}_i^m\right) - \left(\mathcal{S}_j^m - \mathcal{T}_j^m\right) \\ &= \left(\mathcal{S}_i^m - \mathcal{S}_j^m\right) - \left(\mathcal{T}_i^m - \mathcal{T}_j^m\right).\end{aligned} \tag{5}$$

The suspiciousness scores $\mathcal{S}_i^m$ and $\mathcal{S}_j^m$ of each basic block can be computed by an SBFL technique. However, their corresponding false signal components $\mathcal{T}_i^m$ and $\mathcal{T}_j^m$ are not directly available. Instead of computing $\mathcal{T}_i^m$ and $\mathcal{T}_j^m$, we calculate $\mathcal{T}_i^m - \mathcal{T}_j^m$ as a whole to solve the problem. We define the following term

$$\overline{\mathcal{T}_{i,j}^m} \stackrel{\text{def}}{=} \mathcal{T}_i^m - \mathcal{T}_j^m \tag{6}$$

to help explain, whose physical meaning is "the difference in impact brought into $\mathcal{B}_i$ and $\mathcal{B}_j$ by structure bias".

5. For example, $\langle \mathcal{B}_i, \mathcal{B}_j, > \rangle$, $\langle \mathcal{B}_j, \mathcal{B}_k, > \rangle$, and $\langle \mathcal{B}_k, \mathcal{B}_i, > \rangle$ form a ring structure.

TABLE 2: Different Types of Program Versions w.r.t. Two Basic Blocks

	$\{\mathcal{V}^{k1}\}$	$\{\mathcal{V}^{k2}\}$	$\{\mathcal{V}^{k3}\}$	$\{\mathcal{V}^{k4}\}$
Is $\mathcal{B}_i$ a fault? Is $\mathcal{B}_j$ a fault?	no no	yes no	yes yes	no yes
	R_h *w.r.t.* $\langle \mathcal{B}_i, \mathcal{B}_j \rangle$		not included	

To estimate the term $\overline{\mathcal{T}_{i,j}^m}$, we resolve to use the history of fault localization with the program under debugging. Our insight is that "a statement always having high suspiciousness score in all previous program versions has a high chance to be over-estimated to be fault-relevant by a fault localization technique, due to the bias caused by program structure related to it" [47]. With limited data available in practice, we have to approach our goal of capturing the impact by estimating $\overline{\mathcal{T}_{i,j}^m}$ from samples obtained. To do that, we select a set of history versions with respect to $\mathcal{B}_i$ and $\mathcal{B}_j$. The set is thus named *reference set* R_h, with which $\overline{\mathcal{T}_{i,j}^m}$ is calculated as follows:

$$\overline{\mathcal{T}_{i,j}^m} \approx \frac{1}{|R_h|} \cdot \sum_{\mathcal{V}^k \in R_h} \left(\mathcal{T}_i^k - \mathcal{T}_j^k \right), \tag{7}$$

where $|R_h|$ is the size of R_h. This approximation means to approach the difference of impact of structure bias on two basic blocks $\mathcal{B}_i$ and $\mathcal{B}_j$ using the mean difference of impact of structure bias on them across the selected versions in the reference set R_h. Note that for different pairs of basic blocks $\mathcal{B}_i$ and $\mathcal{B}_j$, the constructed R_h can be different. We next explain how we select R_h for basic block pair $\langle \mathcal{B}_i, \mathcal{B}_j \rangle$ to approach $\overline{\mathcal{T}_{i,j}^m}$.

3.3.3 Referencing the History R_h

Table 2 shows all four possible kinds of program versions $\{\mathcal{V}^{k1}\}$ to $\{\mathcal{V}^{k4}\}$ with respect to two basic blocks according to whether or not they are faulty: (1) $\{\mathcal{V}^{k1}\}$ represents the program versions, in which neither $\mathcal{B}_i$ nor $\mathcal{B}_j$ is faulty; (2) $\{\mathcal{V}^{k2}\}$ represents the versions, in which $\mathcal{B}_i$ is faulty while $\mathcal{B}_j$ is not; (3) $\{\mathcal{V}^{k3}\}$ represents the versions, in which both $\mathcal{B}_i$ and $\mathcal{B}_j$ are faulty; (4) and $\{\mathcal{V}^{k4}\}$ represents the versions, in which $\mathcal{B}_i$ is not faulty while $\mathcal{B}_j$ is.

Our goal is to construct a set of program versions, which helps determine the difference of impact of structure bias (see Equation 7) with respect to these two basic blocks (say, $\mathcal{B}_i$ and $\mathcal{B}_j$). To do that, in Table 2, we are interested in the first type of program versions $\{\mathcal{V}^{k1}\}$, because according to our experience the extent of structure bias may be blurred out when a basic block contains the real fault. In consequence, we exclude the program versions $\{\mathcal{V}^{k2}\}$, $\{\mathcal{V}^{k3}\}$, and $\{\mathcal{V}^{k4}\}$, since at least one of $\mathcal{B}_i$ or $\mathcal{B}_j$ is the identified fault of those program versions, and use the program versions $\{\mathcal{V}^{k1}\}$ to construct the reference set R_h, which is given as follows.

$$R_h \; w.r.t. \; \langle \mathcal{B}_i, \mathcal{B}_j \rangle \overset{\text{def}}{=} \Big\{ \mathcal{V}^k \mid \text{both } \mathcal{B}_i \text{ and } \mathcal{B}_j \text{ are not faulty in } \mathcal{V}^k \Big\}, \tag{8}$$

Algorithm 1: Capture the impact $\overline{\mathcal{T}_{i,j}^m}$

Input : set of history versions $\{\mathcal{V}^1, \mathcal{V}^2, \ldots, \mathcal{V}^{m-1}\}$, current version $\mathcal{V}^m$, two basic blocks $\mathcal{B}_i, \mathcal{B}_j$ of $\mathcal{V}^m$
Output : impact $\overline{\mathcal{T}_{i,j}^m}$

```
1  begin
2      R_h ← ∅
3      foreach V_k ∈ {V^1, ..., V^{m-1}} do
4          if both B_i and B_j are blocks in V^k then
5              if neither B_i nor B_j is faulty in V^k then
6                  R_h ← R_h ∪ {V^k}
7      T̄_{i,j}^m ← 0
8      foreach V_k ∈ R_h do
9          T̄_{i,j}^m ← T̄_{i,j}^m + S_i^k − S_j^k
10     T̄_{i,j}^m ← T̄_{i,j}^m / |R_h|
11     return T̄_{i,j}^m
```

where the faultiness of a basic block is determined with respect to history versions. In the evolution between two consecutive history versions, if a basic block is modified in order to fix any failed test cases, we deem it faulty in the former. In such a constructed reference set, we suppose that

$$\lim_{|R_h| \to \infty} \left(\frac{1}{|R_h|} \cdot \sum_{\mathcal{V}^k \in R_h} \mathcal{D}_i^k \right) = \alpha. \tag{9}$$

The left hand side of the above equation calculates the mean of the desired signal for basic block $\mathcal{B}_i$ across all the versions in the reference set R_h, with enough samples. The law of large numbers states that as the size of samples grows, their mean gets closer to the average of the whole population. Since $\mathcal{B}_i$ is not a faulty basic block in R_h, we suppose that the mean of the desired suspiciousness values evaluated for it approaches a constant value α, which is solely related to the fault-localization formula used[6]. As a result, we proceed to deduce Equation 7 as follows, which gets rid of the dependency on α.

$$\begin{aligned}
\overline{\mathcal{T}_{i,j}^m} \;&\approx\; \frac{1}{|R_h|} \cdot \left(\sum_{\mathcal{V}^k \in R_h} \mathcal{S}_i^k - \sum_{\mathcal{V}^k \in R_h} \mathcal{S}_j^k \right) - \\
&\quad\; \frac{1}{|R_h|} \cdot \sum_{\mathcal{V}^k \in R_h} \mathcal{D}_i^k + \frac{1}{|R_h|} \cdot \sum_{\mathcal{V}^k \in R_h} \mathcal{D}_j^k \\
&\approx\; \frac{1}{|R_h|} \cdot \left(\sum_{\mathcal{V}^k \in R_h} \mathcal{S}_i^k - \sum_{\mathcal{V}^k \in R_h} \mathcal{S}_j^k \right) - (\alpha - \alpha) \\
&=\; \frac{1}{|R_h|} \cdot \left(\sum_{\mathcal{V}^k \in R_h} \mathcal{S}_i^k - \sum_{\mathcal{V}^k \in R_h} \mathcal{S}_j^k \right)
\end{aligned}$$

6. Such a value is ever specified as a particular constant (say, 0.5 in [49]) or assumed an unknown constant in existing work [114].

TABLE 3: Descriptive Statistics of Subject Programs

	Subject programs	# of lines	# of versions	# of test cases	Fault type
Siemens suite	print_tokens	194–195	5	4130	seeded
	print_token2	196–200	10	4115	
	replace	241–246	32	5542	
	schedule	151–154	9	2650	
	schedule2	128–130	9	2710	
	tcas	63–67	40	1608	
	tot_info	122–123	23	1052	
Unix utilities*	flex	4002–4035	38	567	
	grep	3198–3466	19	809	
	gzip	1740–2041	32	214	
	sed	1520–3733	30	370	
space		3651–3657	38	13645	real
Defect4J*	Chart	47320–56681	26	2187	
	Closure	36531–62477	133	23601	
	Lang	10977–13933	65	2270	
	Math	5594–46510	106	4369	
	Mockito	2740–5571	38	1363	
	Time	16107–16866	27	4019	

*: programs having history versions

We have shown how to generate a ranked list of suspicious program entities in our model[7]. In the next section, we present how this term $\overline{\mathcal{T}_{i,j}^{m}}$ is calculated with an algorithm with low time complexity and how each corner case is settled.

3.4 Algorithm and Complexity

The process to remove the impact due to structure bias from the observed values is shown in Algorithm 1, which strictly follows the equations in the previous sections.

The loop starting at line 3 constructs the reference set R_h with respect to basic blocks $\mathcal{B}_i$ and $\mathcal{B}_j$. The lines starting at line 8 calculates $\overline{\mathcal{T}_{i,j}^{m}}$. We do not give redundant explanation since it is self-explaining.

The time complexity to calculate $\overline{\mathcal{T}_{i,j}^{m}}$ is $O(m + |R_h|) = O(m)$ because $|R_h| \leq m$, where m is the number of program versions.

Since $O(n^2)$ is the complexity of basic block pairs, the time complexity to sort all basic blocks is finally $O(m \cdot n^2)$, where n is the number of basic blocks.

4 Evaluation

4.1 Experimental Setup

We describe the setup of the experiment in four parts: subject programs, peer techniques, organization of tests, and experiment environment.

4.1.1 Subject Programs

To evaluate our approach, we included the common data set — the Siemens suite of programs, four UNIX utility programs, the program *space*, and Detect4J (JFreeChart, Closure Compiler, Commons Math, Joda-Time, Mockito, and Commons Lang) as our subject programs. The Siemens programs were originally created to support research on data-flow and control-flow test adequacy. Our version of the Siemens suite was downloaded from the Software-artifact Infrastructure Repository (SIR) [17]. All programs of Detect4J are real Java programs containing real faults.

Table 3 shows the descriptive statistics of the subject programs and test pools that we used. The data is with respect to each subject program, including the number of applicable faulty versions, the number of executable statements, and the size of the test pools. In particular, when constructing reference set (according to Algorithm 1), those test cases that were observed pass with more than two program versions were excluded in order to involve less nondeterminism from coincidental correctness. Take print_tokens as an example, it was a program with 194 to 195 lines of executable statements, depending on which subversion. Five versions were used in this experiment, and they shared a test suite consisting of 4130 test cases. Other rows can be interpreted similarly.

Since the Siemens programs were of small sizes and the faults were all manually seeded, we also used five medium-scaled real-life programs with real or seeded faults as additional subjects to strengthen the external validity of our experiment. These programs, also from SIR, including space, flex, grep, gzip and sed, are as shown in Table 3. Each of these programs had one or more versions and each version contained dozens of single faults. We created one faulty program for each single fault, applied the same strategy above to exclude problematic ones, and a total of 285 faulty versions were used as target programs. In our experiment, if any faulty version came with no failure-causing test cases, we did not include it in the experiment. This is because all the experimented subject techniques performed comparisons on faulty programs. We used a UNIX tool, *gcov*, to collect the execution statistics of C language needed for computation. Meanwhile, we used some standard tactics associated with gcov to keep track of the traces of program executions to make the experiment more accurate, even though there were crashing errors during execution.

4.1.2 Subject Techniques

We listed out 34 investigated techniques in Table 4, and our experiment chose nine representative from them [60], [89], [90], [99], [103], [117], namely Op2, Jaccard, Tarantula, Wong2, Wong1, Rogot1, D*[8], Ochiai and Cohen to test. Op2, Jaccard, Tarantula, Wong2, Wong1, Rogot1 were chosen because they were the representations of six equivalent groups [60], [88], [89], [90], [103], [117].

Ochiai and Cohen were chosen from non-grouped techniques [103], [117]. Ochiai was one of the effective fault-localization techniques reported in previous work [4], [80], [113], and Cohen was a complex formula. We chose D* because it was reported outstanding [89], [99]. The selected SBFL formulae were summarized in Table 4.

During software testing, we can get the execution spectrum for a specific test case, namely a tuple $A_i = \langle \mathrm{a_{ef}}, \mathrm{a_{nf}}, \mathrm{a_{ep}}, \mathrm{a_{np}} \rangle$, where $\mathrm{a_{ef}}$ represents the number of failed test cases that execute a basic block $\mathcal{B}_i$, $\mathrm{a_{nf}}$ represents the

7. Note that we set the difference in impact to zero, when there is only one version of the program, $\overline{\mathcal{T}_{i,j}^{m}} = 0$, which means that no history can be referenced.

8. The parameter of D* was chosen as 2. We found that the other parameterization made no significant differences to the empirical observation in our experiment, and thus did not report them in this paper to overload the readers.

TABLE 4: Formulae of Studied SBFL Techniques

group [60], [103]	**formulae**	**representatives in this paper**
ER1	Op1 [60], Op2 [60]	Op2: $\mathrm{a_{ef}} - \frac{\mathrm{a_{ep}}}{P+1}$
ER2	Anderberg [6], Dice [16], Sørensen-Dice [18], Goodman [25], Jaccard [36]	Jaccard: $\frac{\mathrm{a_{ef}}}{F+\mathrm{a_{ep}}}$
ER3	Tarantula [39], q_e [45], CBI inc. [46]	Tarantula: $\frac{\mathrm{a_{ef}}/F}{\mathrm{a_{ef}}/F+\mathrm{a_{ep}}/P}$
ER4	Hamann [31], Euclid [43], Sokal [50], Simple Matching [51], Hamming etc. [60], Rogers & Tanimoto [76], Wong2 [100]	Wong2: $\mathrm{a_{ef}} - \mathrm{a_{ep}}$
ER5	Binary [60], Russel & Rao [78], Wong1 [100]	Wong1: $\mathrm{a_{ef}}$
ER6	Rogot1 [77], Scott [83]	Rogot1: $\frac{\mathrm{a_{ef}}/2}{\mathrm{a_{ef}}+F+\mathrm{a_{ep}}} + \frac{\mathrm{a_{np}}/2}{\mathrm{a_{nf}}+\mathrm{a_{np}}+P}$
non-grouped	Cohen [13], M1 [19], M2 [19], Fleiss [20], Kulczynski1 [50],	Cohen: $\frac{2\mathrm{a_{ef}}\mathrm{a_{np}}-2\mathrm{a_{nf}}\mathrm{a_{ep}}}{(\mathrm{a_{ef}}+\mathrm{a_{ep}})P+(\mathrm{a_{nf}}+\mathrm{a_{np}})F}$
	Kulczynski2 [50], AMPLE [51], Ochiai [63], Arithmetic Mean [77],	Ochiai: $\frac{\mathrm{a_{ef}}}{\sqrt{F(\mathrm{a_{ef}}+\mathrm{a_{ep}})}}$
	D* [99], Wong3 [100], AMPLE2 [103]	D*: $\frac{\mathrm{a_{ef}}^{*}}{\mathrm{a_{ef}}+\mathrm{a_{ep}}}$

number of failed test cases that do not execute the basic block $\mathcal{B}_i$, $\mathrm{a_{ep}}$ represents the number of passed test cases that execute the basic block $\mathcal{B}_i$, and $\mathrm{a_{np}}$ represents the number of passed test cases that do not execute the basic block $\mathcal{B}_i$. Further, we use P and F in these formulae to denote the number of passed test cases and that of failed test cases. By applying each such SBFL formula with these variables collected for each basic block $\mathcal{B}_i$, we computed the suspiciousness score $\mathcal{S}_i$ according to each formula.

4.1.3 Organization of Individual Tests

The experiments in prior studies [15], [60], [61], [117] sometimes classified faulty programs based on the number of known faults in these programs. If there is only one known fault, we name it a *single-fault* scenario. If there is more than one known faults, we name it a *multi-fault* scenario. We also follow this classification of scenarios in this paper.[9] It is worth noting that although many SBFL techniques are developed in the background of single-fault scenario, these techniques have been empirically shown applicable to locate faults in multi-fault scenarios.

We constructed double-fault versions by pair-wisely combining the faults of each subject. If two faults could not coexist, we excluded the generation of the corresponding double-fault version from the experiment. By following the procedure as what we did to synthesize double-fault programs, we also synthesized triple-, quadruple- and pentuple-fault programs for the subjects. For programs containing a sequence of history versions, we simulate the realistic debugging scenario. In the experiment, we used the earliest version to generate the multi-fault subject, and employed the next sequential version to generate a new faulty subject to continue the follow-up tests whenever a fault was located and "fixed". [10] We randomly chose 100 double-fault programs and chose 1000 triple-, 3000 quadruple- and 3000 pentuple-fault programs to simulate multi-fault scenarios. [11] At the same time, we marked the directly affected statement or an adjacent executable statement as faulty when the faulty statement is non-executable (such as [37]). All faulty programs were equipped with a test pool. According to the authors' original intention, the test pool simulates a representative subset of the input domain of the program, so that test suites should be drawn from such a test pool [17].

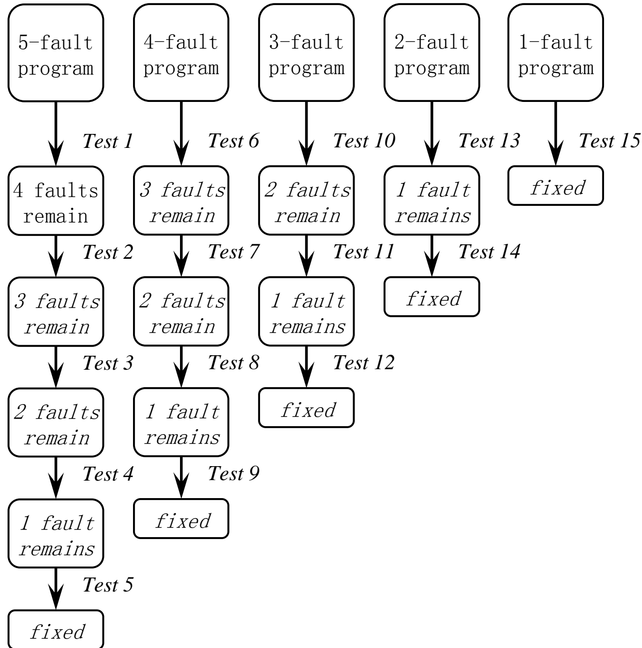
In Section 3.1 we suppose that the identified fault(s) in each faulty program version have been located and subsequently fixed. It is difficult to know how many faults in each fault program. In the experiment, we suppose that the first ranked fault in rank list is the identified fault, and the other faults are not identified. Details of the scheduled tests are listed out in Fig 3. Each "*Test*" in the figure stands for a set of individual tests, each of which is carried out to locate fault in a faulty program. Let us take the first column starting with a block marked "5-fault program" as example to illustrate. It denotes the original program includes 5 faults. By fixing the first fault in the fault localization process "*Test* 1", the program is evolved to having four faults. After the next fault localization process "*Test* 2", it has three faults. Finally, after locating the last fault in the fault localization process "*Test* 5", the program is fixed. The other columns were similarly interpreted.

In the experiment, a total of 15 tests were organized as follows. "*Test* 1", "*Test* 2", "*Test* 3", "*Test* 4", and "*Test* 5" were also classified as belonging to "5-fault" test, "4-fault" test, "3-fault" test, "2-fault" test, and "1-fault" test, respectively, according to how many faults exist in the program under debugging. Specifically, in carrying out "*Test* 1" to locate the first fault in the a "5-fault program", there is no history to reference; so it is also classified as "0-history" test. By the

9. In both scenarios, we investigate the kind of faults, exercising which in a test run is the cause of the failures observed from that test run. In case of a code omission fault, we mark the closest adjacent program entity or directly affected program entity to be "faulty" to continue [113].

10. Take flex for example. We ever enabled faults F_HD_4, F_AA_4, F_HD_1, and F_HD_4 in the version v2.5.1 to generate a pentuple-fault program. During the first test, F_AA_4 was located. We then excluded it and enabled the rest three faults in the version v2.5.2 to generate a triple-fault program to simulate version evolution to continue the rest tests.

11. Different version numbers are treated as different programs and are not cross-referenced.



0-history tests include {*Test* 1, *Test* 6, *Test* 10, *Test* 13, *Test* 15}
1-history tests include {*Test* 2, *Test* 7, *Test* 11, *Test* 14}
2-history tests include {*Test* 3, *Test* 8, *Test* 12}
3-history tests include {*Test* 4, *Test* 9}
4-history tests include {*Test* 5}

1-fault tests include {*Test* 5, *Test* 9, *Test* 12, *Test* 14, *Test* 15}
2-fault tests include {*Test* 4, *Test* 8, *Test* 11, *Test* 13}
3-fault tests include {*Test* 3, *Test* 7, *Test* 10}
4-fault tests include {*Test* 2, *Test* 6}
5-fault tests include {*Test* 1}

Fig. 3: Experiment Setup for Multi-fault Tests

same token, we classified "*Test* 2", "*Test* 3", "*Test* 4", and "*Test* 5" to be "1-history", "2-history", "3-history", and "4-history" test, respectively, according to how many history versions can be referenced in carrying out the tests. To simulate a more realistic scenario, each "Test" only employed 80% test cases, and there will be a 5% difference of test case between adjacent tests. For example, we randomly choose 80% of the corresponding test cases in "*Test* 1", and choose 5% of the test cases from the remaining test cases to replace the test cases of "*Test* 1" in"*Test* 2".

Based on the execution profile files generated by the logging tool *gcov*, we collected the passed and failed test cases for each program in Table 3, then calculated the suspiciousness score for each basic block using each SBFL techniques stated in Table 4 as well as enabling Delta4Ts on each such SBFL technique, finally compared their accuracy in locating faults. In practice, it could be very difficult to identify the history of a block, and we use historical slicing [81], [82] to help us complete this experiment. In our evaluation, we carried out 576,000 individual tests, i.e., $(100\times2+1000\times3+3000\times4+3000\times5)\times9\times2$, in total. Note that there was no need to count tests with 0-history or carry out tests on 1-fault program since our model degenerates to the underlying SBFL technique, resulting in no accuracy change, when there was no history for reference.

4.1.4 Experiment Environment

The experiment was carried out on a ThinkCentre m8300t desktop with 4-core Intel(R) Core(TM) i7-2600 (3.40GHz) processors, 16GB physical memory and 500GB hard disk equipped Ubuntu13.10 operating system with the kernel version of 3.11.0-19-generic. We used the tools of gcc 4.6.4 and gcov 4.6.4 to construct our experiment platform. The testbed, scripts, and the prototype toolset are available on https://github.com/lvshuiyiqing/Delta4Ts-benchmark.

4.2 Performance Measurement

In this section, we review some accuracy operators which denote the cost to locate one fault. Meanwhile, we also review some accuracy metrics proposed to investigate and measure the accuracy of existing fault localization techniques.

4.2.1 Accuracy Metrics

Each of the studied techniques generates a ranked list of all the executable entities. The Eff metric is a popular measure to assess the fault-localization quality of a ranked list. Eff metric denotes that how many executable entities need to be checked.

$$\mathsf{Eff}(T) = \text{ the rank of the faulty statement} \tag{10}$$

The lower the value is, the higher the accuracy of the fault localization technique is.

However, different programs may contain different numbers of basic blocks, which makes direct comparisons using Eff unsuitable. We further use other popular metrics Exp [113] to analyze the data of our experiments. Exp denotes that how many percent of the executable entities need to be checked.

$$\mathsf{Exp}(T) = \frac{\mathsf{Eff}(T)}{\text{number of all executable statements}} \times 100\% \tag{11}$$

Exp assesses the relative cost of using the generated ranked list to locate a fault. Both Eff and Exp have been discussed in existing work as neither of them accurately reflects the cost of a programmer debugging programs in reality [42]. On the other hand, to the best of our knowledge, no known metric that can accurately reflect the debugging cost is discovered yet.

In spite of the known issues with Eff and Exp, they are still popularly used in evaluating the accuracy of SBFL techniques. Apart from them, we also use the Sav metric to measure the improvement on the Eff and Exp metrics made by Delta4Ts. Sav metric denotes how much checking cost is saved before and after using our model. Sav (*Saved* in short) [12] is defined as follows:

$$\mathsf{Sav}(T) = \frac{\mathsf{Eff}(T)\ - \mathsf{Eff}(T')}{Avg\{\mathsf{Eff}(T), \mathsf{Eff}(T')\}} \times 100\%. \tag{12}$$

Note that, in this paper, Eff(T) and Eff(T′) denote how many executable entities need to be checked before the first fault is located. For giving an SBFL and a program P with m history versions, we use the SBFL to compute Eff(T) on P,

12. Note that we use one basic SBFL and Delta4Ts to generate the corresponding ranked lists of all the executable entities on one subject, and then obtain Eff(T) and Eff(T′).

and use Delta4Ts to compute Eff(T′) on P with m history versions. Then, we compare Eff(T) and Eff(T′) to obtain $\mathsf{Sav}(T)$. $\mathsf{Sav}(T) > 0$ denotes that Delta4Ts can improve the effectiveness of SBFL, $\mathsf{Sav}(T) < 0$ denotes that Delta4Ts can deteriorate the effectiveness of SBFL, and $\mathsf{Sav}(T) = 0$ denotes that Delta4Ts does not affect the effectiveness of SBFL.

At the same time, the rank of the faulty statement in the resultant list also relies on the tie-breaking strategies and examination affordability which are introduced int the next two subsections.

4.2.2 Tie-breaking Strategies

It is possible that some non-faulty entities have identical suspiciousness score as the faulty entity, and it becomes necessary to determine the order of the entities sharing identical suspiciousness score.

We note that the widely-used fractional ranking scheme (aka, the "Avg" scheme [101]) is used throughout the experiment to assign the faulty statement with a rank when reporting the analysis results. However, different tie-breaking strategies may result in the data analysis of fault localization experiment. For instance, the *Min-* strategy assumes the best tie-breaking result that the faulty statement is the first entity to be examined among all the entities with identical suspiciousness score. On the contrary, the *Max-* strategy assumes the worst tie-breaking result that the faulty statement is the last entity to be examined among all the entities with identical suspiciousness score [47], [100].

In the rest of this experiment, we refer to the above two fault-localization manners as *Min-* and *Max-*, which were used in Section 4.3.3 to analyze the data. Their alternative way *Avg-*, which assumes that the faulty statement is the middle one to be examined among all the entities with identical suspiciousness score, is used throughout the whole data analysis unless stated otherwise.

4.2.3 Examination Affordability

Evaluating the accuracy of the fault localization techniques is an important way to compare the effectiveness of different fault localization techniques. Jones et al. [38] used a score to evaluate the effectiveness of these techniques, and the score is defined as the percentage of code that needs not be examined to find one fault entity. For example, given one program and a test suite, one fault localization technique T generates a rank list, which has m elements, and the n-th element is the fault of the given program. Namely, the accuracy of T is $(m - n)/m$ for the given program and test suite. The accuracy metric [38] has become the standard way to evaluate the effectiveness of SBFL techniques.

Evaluating the practical accuracy of the fault localization techniques needs to consider the effort a developer can afford to spend [42], [85]. In practice, developers may only care about the top portion of the rank list generated by a fault localization technique T. If the faulty entity is ranked after the top selected (say, k) entities, developers may give up the scanning after checking the top k entities and switch to use other debugging trials. As a result, the attempt to locate any fault using the rank list will be considered in vain. When handling programs of a larger scale, a developer may be more tolerant to the accuracy of a fault-localization tool and wants to check more (say, the top $k\%$ or the top k‰) suspicious candidates, than usual before giving up [117]. If a program entity appears beyond the top ranked region (either top k, top $k\%$ or top k‰) in the ranked list, its rank is calculated as the length of the whole list; in other words, the practical accuracy in such a case is deemed zero.

In the rest of this experiment, we refer to the above fault-localization manners as *Top-k*, *Top-k%*, and *Top-k‰*, which are used in Section 4.3.3; and refer to their alternative way (to check the entire list) as *Whole list*, which is used throughout the whole experiment.

4.3 Data Analysis and Results

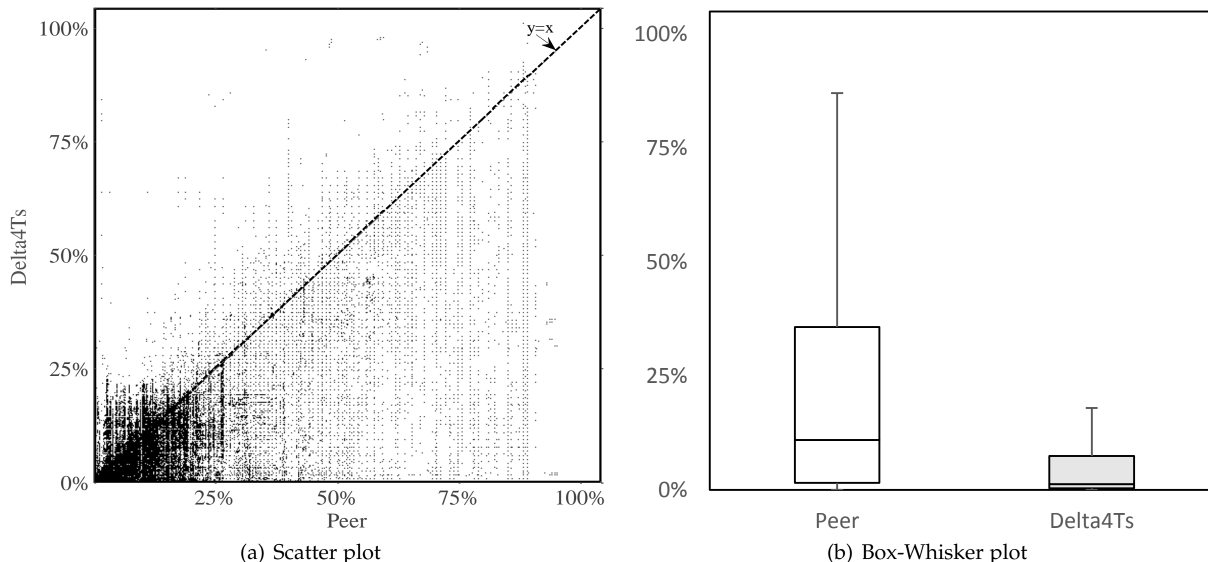
In this section, we visualize the results to present the accuracy improvement brought by Delta4Ts.

4.3.1 Data Visualization and Statistics

Table 11 in the appendix summarizes the change in Eff from applying each peer SBFL technique to applying Delta4Ts on each subject. Let us take the first section as example. "Op2 on print_tokens: 77→77, 86→86, 96→17, 95→11, 30→3." denotes that the Eff of Op2 on the program print_tokens contains five faults, each of which are located by examining 77, 86, 96, 95, and 30 basic blocks using the technique Op2, while they can be located by examining 77, 86, 17, 11, and 3 basic blocks using Delta4Ts to drive Op2, respectively.

We observe from the table that in most of the cases, the faults are more accurately located after applying Delta4Ts. Moreover, the exceptions mainly appear with the first fault located in each test, since there is no history to reference to locate the first fault, our model generates identical output as its peer technique used for comparison. There are many data even we have summarized them. Therefore, we resolve to analyze them using statistics.

Fig. 4 shows the overall accuracy of Delta4Ts and the aggregated results of the peer SBFL techniques. The scatter plot (see Fig. 4(a)) shows the overall fault-localization accuracy on all the programs and all the SBFL techniques by applying the Peer techniques directly and by applying Delta4Ts. In this plot, the x-axis indicates the accuracy (in Exp) of a $Peer$ technique, and the y-axis indicates the accuracy (in Exp) of Delta4Ts. There are 288,000 data points in Fig. 4(a). Each point in the plot maps to a comparison result of $Peer$ and Delta4Ts with respect to an SBFL technique on a subject program. In the figure, we also draw the line $y = x$, which separates the tests, on which our model outperforms the $Peer$ technique, from the tests, on which $Peer$ outperforms Delta4Ts. We observe that most scatter points are below the line $y = x$. It indicates that Delta4Ts can outperform a $Peer$ technique in fault-localization accuracy in most cases. More importantly, Delta4Ts embeds a systematic method to compare the desired suspiciousness scores of the program entities to be ranked. At the same time, we also notice that there are still some counter-example points (those above the line $y = x$) in Fig. 4(a), which shows that Delta4Ts still has rooms for further research. A close look at the data disclosed the reason that those faulty blocks were faulty in some history versions. In consequence, they received relative high suspiciousness scores in those versions. Since we marked the faultiness of a basic block by whether



(a) Scatter plot (b) Box-Whisker plot

Fig. 4: Aggregated Accuracy (in Exp) for All Techniques on All Programs.

TABLE 5: Accuracy Improvement (in Sav) for Each Technique on Each Program.

	Op2	Jaccard	Tarantula	Wong2	Wong1	Rogot1	Cohen	Ochiai	D*	*Avg*
print_tokens	56.7%	52.9%	82.0%	63.2%	14.0%	60.6%	61.0%	45.2%	33.4%	52.1%
print_token2	114.2%	41.9%	54.7%	58.6%	72.7%	41.5%	38.7%	71.2%	84.7%	64.2%
replace	76.5%	37.9%	62.2%	37.9%	38.5%	24.3%	37.2%	46.2%	50.2%	45.6%
schedule	62.5%	13.0%	37.7%	15.3%	18.6%	12.1%	12.6%	11.7%	25.2%	23.2%
schedule2	9.1%	35.6%	15.7%	24.3%	36.7%	35.3%	36.0%	40.4%	65.0%	33.1%
tcas	47.8%	37.6%	46.8%	19.3%	42.1%	33.5%	36.9%	47.7%	19.3%	36.8%
tot_info	15.4%	20.1%	12.7%	56.2%	41.4%	27.1%	18.7%	35.3%	74.5%	33.5%
space	122.9%	5.2%	70.9%	10.3%	11.4%	5.1%	4.9%	11.0%	15.1%	28.54%
flex	24.5%	19.6%	26.2%	19.1%	26.2%	19.6%	19.6%	19.6%	26.2%	22.3%
sed	113.2%	10.4%	67.9%	9.3%	14.8%	7.0%	8.1%	15.2%	20.8%	29.6%
gzip	46.8%	1.8%	68.4%	3.5%	55.7%	2.0%	2.1%	4.7%	58.2%	27.0%
grep	42.6%	12.7%	24.6%	10.7%	23.1%	9.3%	13.4%	29.3%	26.4%	21.3%
Avg	61.0%	24.1%	47.5%	27.3%	32.9%	23.1%	24.1%	31.5%	41.6%	34.8%

it was modified during program evolution (recalling the explanation to Equation 8), these fault-relevant basic blocks were marked non-faulty in history versions. And in the current version where they resulted in failures, their suspicious score are adjusted too much in the opposite direction. We further use plot (b) to give a statistical comparison.

In the box-whisker plot (see Fig. 4(b)), we use two columns to show the accuracy of $Peer$ and Delta4Ts, where $Peer$ stands for the overall accuracy of peer SBFL techniques. For each column, the horizontal line in the box indicates the median value of the accuracy to locate faults in each faulty version of a specific program. The bottom of the box corresponds to the 25% percentile, while the top of the box corresponds to the 75% percentile of the collected data. The upper whiskey shows the maximum Exp within 1.5 IQR [5] of the upper quartile, while the lower whiskey shows the minimum Exp with 1.5 IQR of the lower quartile. In Fig. 4(b), the y-axis indicates the accuracy (in Exp) of Delta4Ts. The lower the box and the lines are, the better the the corresponding method is.

Let us take Peer for discussion first. The lower whisker shows that the minimum Exp to locate a fault is 0.03%. The upper whiskey shows that the maximum Exp to locate a fault is 86.0%. The bottom and top of the box show the 25% and 75% percentiles for Exp are 1.5% and 35.3%, respectively. The horizontal line in the box indicates that the median Exp is 11.0%. Using Delta4Ts, the minimum, median and maximum Exp are 0.02%, 1.3%, and 18.0%, respectively. The 25% and 75% percentiles for Exp are 0.3% and 7.4%, respectively. There are some counter-example points above the line $y = x$ in Fig. 4(a), but the results of Fig. 4(b) indicate that our model can outperform $Peer$ in fault-localization accuracy in most cases. More importantly, the accuracy of Delta4Ts is much higher than the accuracy of $Peer$ in most cases.

We further analyze the fault localization accuracy improvement made by Delta4Ts on each individual program and each individual SBFL technique. Table 5 summarizes the results. Let us take the cell for the Op2 column and the print_tokens row as example. The value of 56.7% indicates that the Eff values of Tarantula on the program print_tokens are on average saved by 56.7% after using Delta4Ts based on the metric Sav. Other cells in the table can be interpreted similarly. Moreover, by applying Delta4Ts on a specific technique across all the programs, the mean Sav ranges from 23.1% to 61.0%. Similarly, by applying Delta4Ts on a

specific program across all the SBFL techniques, the mean Sav ranges from 21.3% to 64.2%. On average, 34.8% saving can be made by Delta4Ts to locate a fault. It indicates Delta4Ts can improve the effectiveness of fault localization techniques on average.

We have examined all the cells in Table 5, and find the mean Sav ranges from 1.8% to 122.9%, which indicates Delta4Ts can improve every technique on every program. At the same time, we also notice that the value of the mean Sav is 1.8% on Tarantula and gzip, which indicates Delta4Ts has little effect on Tarantula and gzip. A close look at multi-fault versions discloses the reason that a small number of statements are assigned suspicious values and have little intersection in different versions, which is difficult to obtain the impact of program structure by a statistical method. We have presented in Section 4.1.2 that some SBFL techniques are included in the experiment because they have shown to exhibit theoretical relations in terms of fault localization accuracy. In particular, Op2 and Wong1 are the representatives from the corresponding equivalent groups (ER1 and ER5) which are the maximal (i.e., most accurate) formulae in the theoretical study on SBFL formulae [103]. Previous empirical studies [89], [117] have shown that Op2 is more accurate than Wong1. From Table 5, we find that overall speaking, the improvement made by Delta4Ts on Op2 and Wong1 are 61.0% and 32.9%, respectively.

4.3.2 Effectiveness Stability

To verify whether the results observed in the last two subsections can be statistically meaningful, we adopt the Wilcoxon signed rank test method.

The Wilcoxon signed rank test is a nonparametric test for two populations when the observations are paired. The p-value of the Wilcoxon signed rank test is used to measure how much the evaluation biases between the observations in the two samples and judge whether the null hypothesis H_0 can be rejected or not. Here, a p-value less than a chosen significance level (e.g., 0.05) indicates the rejection of the null hypothesis H_0, otherwise a failure to reject the null hypothesis H_0 at the chosen significance level. We further reference the Cliff's Delta [12] method to help explain the results. The Cliff's Delta is a measurement to how often the values in one distribution are larger than the values in another distribution. Crucially, it does not require any assumptions about the shape or spread of the two distributions. The d value of Cliff's Delta is between -1 and 1. Here, $d = 0$ indicates that there is no observable difference between the two distribution of samples, while $d = 1$ (-1) indicates that there is the strongest positive (negative) difference. Usually, according to the value of d, the statistical significance of Cliff's Delta is marked as trivial ($|d| < 0.147$), small ($0.147 \leq |d| < 0.33$), moderate ($0.33 \leq |d| < 0.474$) or large ($|d| \geq 0.474$) label.

Table 6 and 7 present the results of the above hypothesis testing. In these two tables, the "Wilcoxon (p)" column shows the results of Wilcoxon signed rank test at the significance level of 0.05, and the "Cliff's Delta (d)" column shows the results of Cliff's Delta test. In each table, we observe that the p values of Wilcoxon signed rank test are always less than 0.05 (in the range of $[4.2 \times 10^{-32}, 1.9 \times 10^{-21}]$ actually), and the d values of Cliff's Delta are always above zero (in the range $[0.27, 0.66]$ actually). It means that the null hypothesis is always rejected at the significance level of 0.05 and a positive difference always exist. To further analyze to what extent Delta4Ts can improve SBFL on the subject programs, we compute the median statistical indexes and their standard deviations for the tests. In Table 6 and Table 7, we observe that (i) the null hypothesis can be always rejected, which means that the improvements by Delta4Ts are determined statistically significant, and (ii) although the median improvements are always positive (always more than 17.1% for different techniques and more than 17.0% for different programs), the relatively large standard deviation should be studied carefully (may reach 50.8% for some techniques and reach 74.5% for some programs).

In summary, we found that improvements in a statistically meaningful way had been observed on the fault-localization accuracies of (i) the nine individual techniques across all subject programs and (2) on the eleven individual subjects across all the techniques, when Delta4Ts was applied.

We next investigate the factors affecting the effectiveness of Delta4Ts in Section 4.3.3.

4.3.3 Impact Factors on Effectiveness

In this section, we analyze the impact of different numbers of history versions on the accuracy of Delta4Ts. Fig. 5[13] shows the box-whisker plot in terms of both Eff and Sav on programs having different numbers of history versions. Fig. 5(a) shows the Eff results of programs with different number of history versions, which are classified into Peer and Delta4Ts, representing before and after using Delta4Ts. In each group, there are four columns (1-history, 2-history, 3-history, and 4-history). Let us take 1-history of Peer for discussion first. The lower whisker shows that the minimum Eff to locate a fault is 1. The upper whiskey shows that the maximum Eff to locate a fault is 191. The bottom and top of the box shows the 25% and 75% percentiles for Eff are 11 and 83, respectively. The horizontal line in the box indicates that the median Eff is 30. The minimum, median, and maximum Eff of 1-history of Delta4Ts are 1, 21, and 144.5, respectively. The 25% and 75% percentiles for Eff are 7 and 62, respectively. In order to clearly show the effectiveness of Delta4Ts, Fig. 5(b) shows the Sav results. The minimum, median, and maximum Sav of 1-history are -90.0%, 22.0%, and 174.0%, respectively. The 25% and 75% percentiles for Eff are 9.1% and 75.2%, respectively. It indicates that Delta4Ts likely improves the effectiveness of SBFL with 1 history version. At the same time, Delta4Ts can always improve the effectiveness of SBFL with 2, 3, and 4 history versions, respectively. Fig. 5 shows that Delta4Ts can improve the effectiveness of SBFL with different history versions. The general trend from these columns is that the accuracy of Delta4Ts increases as more history versions can be used.

We are interested in the impacts of number of faults on the accuracy improvement of Delta4Ts. Fig. 6(a) and

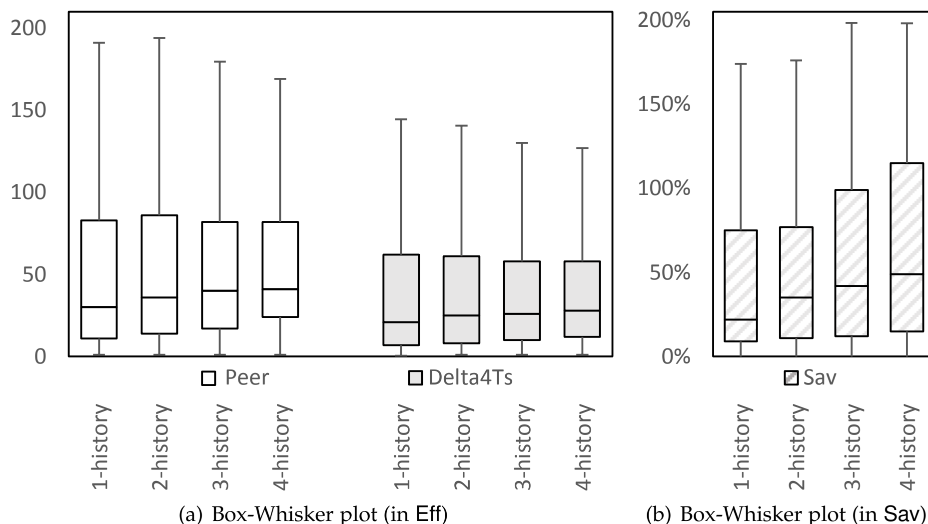
13. The median value trend from this plot increases as more history versions can be used. Usually, the more faults one program contains, the easier it is to locate the first fault.

TABLE 6: Accuracy Improvement (in Sav) w.r.t. Different Techniques.

	Op2	Jaccard	Tarantula	Wong2	Wong1	Rogot1	Cohen	Ochiai	D*
Wilcoxon (p)	2.1e-31	9.1e-32	4.1e-31	7.9e-32	1.3e-24	6.2e-25	3.3e-29	1.2e-30	4.2e-32
Cliff's Delta (d)	0.49	0.51	0.52	0.52	0.43	0.48	0.57	0.51	0.41
median	55.0%	22.5%	40.6%	24.1%	26.7%	17.1%	21.4%	27.7%	36.6%
stdev	50.8%	26.1%	38.7%	18.6%	33.2%	39.7%	47.0%	40.6%	48.8%

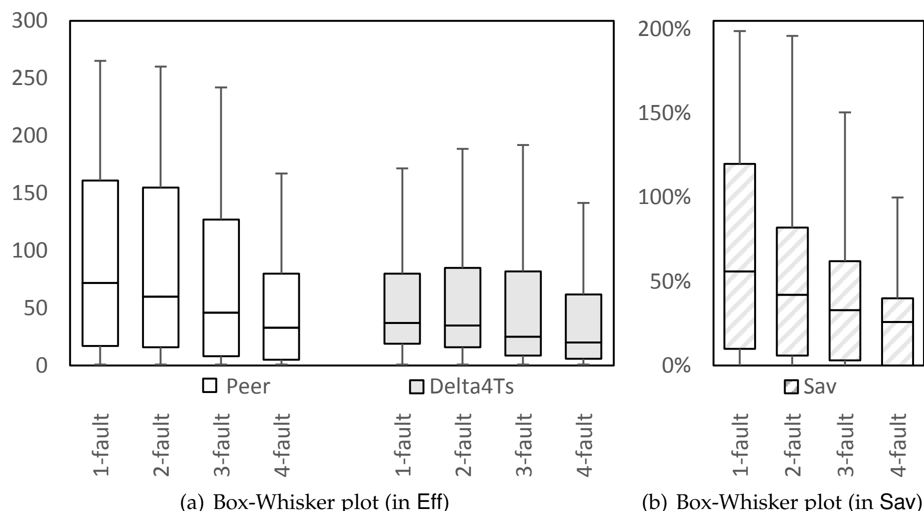
TABLE 7: Accuracy Improvement (in Sav) w.r.t. Different Programs.

	print_tokens	print_token2	replace	schedule	schedule2	tcas	tot_info	flex	sed	gzip	grep	space
Wilcoxon (p)	1.6e-24	5.8e-25	1.9e-21	6.2e-28	4.3e-27	9.5e-26	6.4e-29	7.4e-30	5.3e-27	2.6e-25	4.0e-26	2.1e-28
Cliff's Delta (d)	0.66	0.58	0.49	0.27	0.43	0.51	0.49	0.54	0.49	0.59	0.41	0.51
median	44.8%	50.5%	45.5%	19.1%	31.9%	28.5%	21.5%	20.1%	29.3%	19.7%	17.0%	23.1%
stdev	37.5%	45.7%	57.5%	36.8%	21.8%	11.9%	35.6%	48.3%	74.5%	59.8%	18.2%	56.3%



(a) Box-Whisker plot (in Eff) (b) Box-Whisker plot (in Sav)

Fig. 5: Impact of History Number on Accuracy Improvement

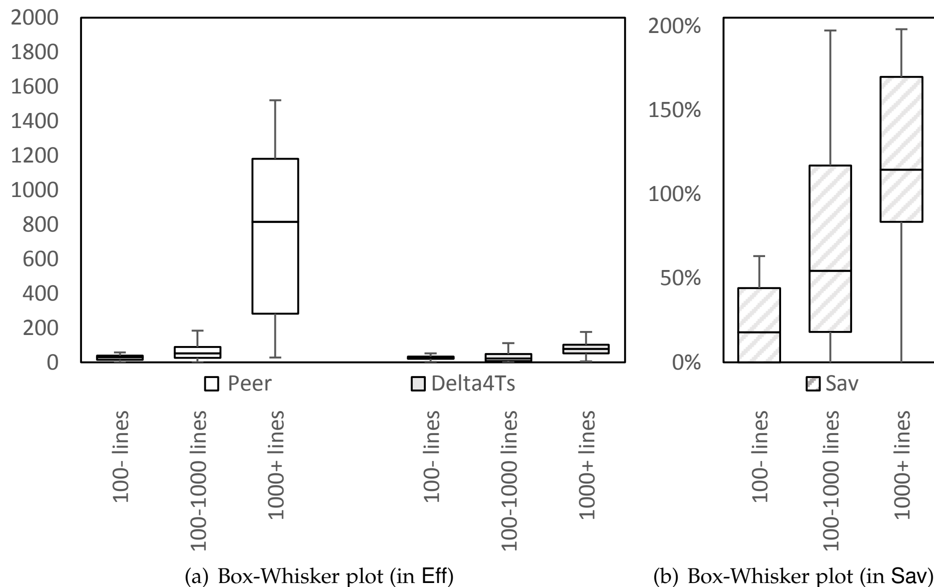


(a) Box-Whisker plot (in Eff) (b) Box-Whisker plot (in Sav)

Fig. 6: Impact of Number of Faults on Accuracy Improvement

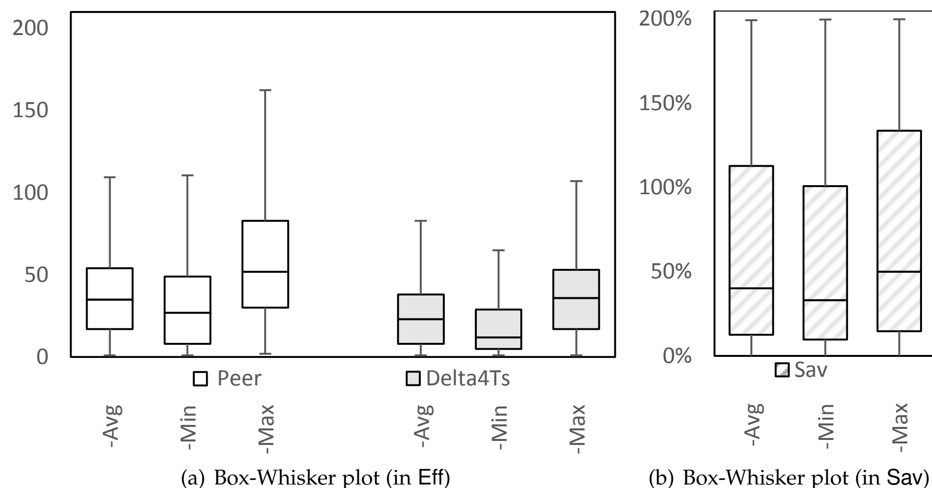
(b)[14] summarize the results in terms of Eff and Sav in box-whisker plot on programs having different numbers of faults. Fig. 6(a) shows the Eff results of programs with different numbers of faults, which are classified into Peer and Delta4Ts, representing before and after using Delta4Ts. In

14. Our model does not improve the programs with 0 history version, and we do not show the results of 5-fault programs which has no history versions in Fig. 6.



(a) Box-Whisker plot (in Eff) (b) Box-Whisker plot (in Sav)

Fig. 7: Impact of Program Scale on Accuracy Improvement



(a) Box-Whisker plot (in Eff) (b) Box-Whisker plot (in Sav)

Fig. 8: Impact of Tie-breaking Strategy on Accuracy Improvement

each group, there are four columns (1-fault, 2-fault, 3-fault, and 4-fault). Let us take 1-fault of Peer for discussion first. The lower whisker shows that the minimum Eff to locate a fault is 1. The upper whiskey shows that the maximum Eff to locate a fault is 161. The bottom and top of the box show the 25% and 75% percentiles for Eff are 31 and 83, respectively. The horizontal line in the box indicates that the median Eff is 51. The minimum, median, and maximum Eff of 1-fault of Delta4Ts are 1, 30, and 136.5, respectively. The 25% and 75% percentiles for Eff are 19 and 62, respectively. In order to clearly show the effectiveness of our model, Fig. 6(b) shows the Sav results. The minimum, median, and maximum Sav of 1-fault are -146.9%, 73.7%, and 198.1%, respectively. The 25% and 75% percentiles for Eff are 14.2% and 121.6%, respectively. It indicates that Delta4Ts likely improves the effectiveness of SBFL on the programs with 1 fault. At the same time, Delta4Ts can always improve the effectiveness of SBFL on the programs with 2, 3 and 4 faults, respectively. From Fig. 6, we observe that the number of faults is not a significant factor affecting the accuracy improvement made by Delta4Ts.

Keller et al. [41] reported that the scale of programs might affect the accuracy of SBFL observably. We thus also analyze the impact of different scales of programs on the accuracy of Delta4Ts. Fig. 7 shows the Exp and Sav results of these three scale-categories in box-whisker plot. Fig. 7(a) shows the Eff results of programs with different scales, which are classified into Peer and Delta4Ts, representing before and after using Delta4Ts. In each group, the results are classified into three mutually exclusive scale-categories ($100-$ lines, $100-1000$ lines and $1000+$ lines), according to the scale of programs. Let us take "1000+ lines" of Peer for discussion first. The lower whisker shows that the minimum Eff to locate a fault is 29. The upper whiskey shows that the maximum Eff to locate a fault is 1521. The bottom and top of the box show the 25% and 75% percentiles for Eff are

TABLE 8: Impact of Examination Affordability on Accuracy Improvement

	# and % of faults located		Avg. of rank	
	Peer	Delta4Ts	Peer	Delta4Ts
Top-5%	123,866 (43.0%)	170,125 (59.1%)	15.1	15.5
Top-5	96,015 (33.3%)	131,211 (45.6%)	2.8	2.6
Top-5‰	23,133 (8.0%)	29,971 (10.4%)	4.5	4.2

283 and 1181, respectively. The horizontal line in the box indicates that the median Eff is 815. The minimum, median, and maximum Eff of "1000+ lines" of Delta4Ts are 6, 79, and 178, respectively. The 25% and 75% percentiles for Eff are 53 and 103, respectively. In order to clearly show the effectiveness of Delta4Ts, Fig. 7(b) shows the Sav results. The minimum, median, and maximum Sav of "1000+ lines" are -45.7%, 114.7%, and 198.1%, respectively. The 25% and 75% percentiles for Eff are 83.5% and 169.7%, respectively. It indicates that Delta4Ts likely improves the effectiveness of SBFL on the programs with 1000+ line. At the same time, Delta4Ts can always improve the effectiveness of SBFL on the programs with 100− lines and 100−1000 lines, respectively. Fig. 7 shows that Delta4Ts is more effective for larger programs.

We further analyze the data using other metrics. Similar to the Avg tie-breaking strategy, the Max and the Min tie-breaking strategies assume that the target statement is the last and first one in a tie group, in which manner, the Max expense and Min expense to locate a fault are measured, respectively. Fig. 8 shows the Exp and Sav results of using these three ranking schemes, respectively. Fig. 8(a) shows the Eff results of programs with different ranking schemes, which are classified into Peer and Delta4Ts, representing before and after using Delta4Ts. In each group, the results are classified into three mutually exclusive scheme-categories (Avg, Max, and Min), according to the ranking schemes. Let us take Avg of Peer for discussion first. In Fig. 8(a), the lower whisker shows that the minimum Eff to locate a fault is 1. The upper whiskey shows that the maximum Eff to locate a fault is 109.5. The bottom and top of the box show the 25% and 75% percentiles for Eff are 17 and 54, respectively. The horizontal line in the box indicates that the median Eff is 35. The minimum, median, and maximum Eff of Avg of Delta4Ts are 1, 23, and 83, respectively. The 25% and 75% percentiles for Eff are 8 and 38, respectively. In order to clearly show the effectiveness of our model, Fig. 8(b) shows the Sav results. The minimum, median, and maximum for Sav are -137.6%, 40.3%, and 198.1%, respectively. The 25% and 75% percentiles for Sav are 12.6% and 112.7%, respectively. It indicates Delta4Ts can improve the effectiveness of SBFL using Avg ranking scheme. At the same time, Delta4Ts can also improve the effectiveness of SBFL using Max and Min ranking schemes, respectively. We observe that using different ranking schemes does not produce significant effects on the accuracy improvement made by Delta4Ts.

In practice, developers may only check about the top portion of the fault localization recommendation generated by a technique. Thus, apart from measuring Delta4Ts using the "Whole list", we also analyze the data using the Top-5, Top-5%, and Top-5‰ metrics, which mean to examine whether the faulty statement is included in the top 5 candidates, and the top 5% and top 5‰ of the ranked list generated by a technique. Table8 shows that before and after using Delta4Ts, how many versions are effective and what the average Eff of effective versions is, using the Top-5, Top-5%, and Top-5‰ metrics. Let us take the row of Top-5% for discussion first. The data shows that in 123,866 (43.0% of all the 288,000) tests, the faulty basic block is assigned a high rank by Peer and not beyond the top 5% suspicious blocks, and the rank given to them is 15.1 on average. At the same time, Delta4Ts can recognize faults in 170,125 (59.1% of all) tests with the same precision, and the rank given to them is 15.5 on average. The other lines can be similarly explained. Since the ranks on average (the column "Avg. of rank") cannot be directly compared, we concentrate on the number and percentage of faults located (the column "# and % of faults located") in Table 8. From Table 8, we observe that whatever affordability alternatives is chosen, Delta4Ts is always effective using different metrics to improve the accuracy of SBFL techniques.

TABLE 9: Summary of Impact Factor Observations

	Alternatives	Observation
No. of history versions	1, 2, 3, and 4	effective for each number, and better with more history
No. of faults	1, 2, 3, and 4	effective for each number, and better with fewer faults
Program scale	100-, 100-1k, and 1k+	effective for each range, and better on larger programs
Tile-breaking	-Avg, -Min, and -Max	effective for each strategy
Examination affordability	Whole list, Top-5, Top-5%, and Top-5‰	effective for each range

We have reported five categories of data analysis to evaluate Delta4Ts. Table 9 summarizes the major findings. It summarizes the effect of the number of history versions, number of faults in each program, program scale, and ranking scheme on the accuracy improvement made by Delta4Ts on SBFL techniques. In particular, Delta4Ts is more effective when applying it to programs larger in scale and having more history versions for reference.

4.4 Effectiveness on large scale real programs

Although we have analyzed the impact of different factors on Delta4Ts over 12 common benchmarks and the experiment shows Delta4Ts can improve the effectiveness of SBFL, most faults of these benchmarks are seeded. In this section, we further analyze the effectiveness of Delta4Ts on large scale real programs – Defect4J.

In the experiment, most classes were not faulty. We did not know which classes caused a test case to fail in a JUnit test class. For debugging purpose, we follow other related work of SBFL [89] to include the executable statements in all these application classes as potential candidates of faulty statements. We checkout a buggy source code version from Detects4J and mark the code version number based on the

TABLE 10: Accuracy Improvement (in Sav) for Each Technique on Each Program.

	Sequences	Op2	Jaccard	Tarantula	Wong2	Wong1	Rogot1	Cohen	Ochiai	D*	*Avg*
Closure	half	24.9%	17.6%	20.6%	41.6%	22.7%	30.6%	18.5%	18.3%	15.6%	23.4%
	all	27.1%	26.2%	28.3%	52.7%	32.3%	33.3%	25.2%	33.4%	23.7%	31.3%
chart	half	11.6%	10.9%	17.7%	14.2%	20.6%	24.5%	17.4%	12.1%	14.0%	15.9%
	all	26.4%	23.6%	24.2%	38.6%	39.8%	32.7%	29.9%	27.5%	19.3%	29.1%
Lang	half	23.5%	22.8%	13.8%	10.1%	23.2%	27.6%	22.3%	23.8%	18.6%	20.6%
	all	26.7%	29.4%	24.5%	45.2%	27.6%	28.4%	30.9%	30.8%	19.3%	29.2%
Math	half	13.3%	13.8%	11.3%	29.2%	21.6%	11.6%	8.3%	9.7%	13.0%	14.7%
	all	32.0%	21.1%	28.1%	40.6%	42.0%	17.4%	18.9%	30.4%	21.6%	28.0%
Mockito	half	17.4%	4.6%	7.6%	27.5%	10.5%	20.0%	15.4%	5.2%	6.5%	12.7%
	all	27.7%	14.5%	16.9%	50.1%	36.4%	27.2%	35.1%	13.9%	15.5%	26.4%
Time	half	36.7%	19.3%	39.5%	41.6%	38.8%	43.4%	22.1%	27.4%	23.2%	32.4%
	all	43.7%	22.1%	43.3%	65.8%	45.5%	47.4%	24.2%	33.7%	31.5%	39.7%
Avg	half	21.2%	14.8%	18.4%	27.4%	22.9%	26.3%	17.3%	16.1%	15.1%	19.9%
	all	30.6%	22.8%	27.6%	48.8%	37.3%	31.1%	27.4%	28.3%	21.8%	30.6%

bug number. We randomly select 80% test cases from the corresponding test suite for each version. In Delta4Ts, we use all history versions to capture the impact of structure bias. In section 4.3.3, we have analyzed the impact of different history number on the accuracy improvement of SBFL, but we always used all history versions.

In this section, we further analyze whether different sequences of versions have an impact on the accuracy improvement, and we compare the accuracy improvement by using all history versions and half of all history versions. Table 10 summarizes the results of the accuracy improvement by using all history versions and half of all history versions, respectively. In Table 10, "half" denotes half of all history versions, and "all" denotes all history versions. Let us take the cell for the Op2 column and the print_tokens row as example. The value of 24.9% indicates that Eff values of Op2 on the program Closure are on average saved by 24.9% after using Delta4Ts with half of all history versions based on the metric Sav. The value of 27.1% indicates that Eff values of Op2 on the program Closureprint_tokens are on average saved by 56.7% after using Delta4Ts with all history versions based on the metric Sav. Other cells in the table can be interpreted similarly. Moreover, while Delta4Ts can obtain the mean value of Sav ranging from 13.9% to 65.8% with all history versions as inputs, the mean value ranges from 4.6% to 43.4% given only a half number of history versions.

On average, 19.9% saving can be made by Delta4Ts with half of all versions to locate a fault, and 30.6% saving can be made by Delta4Ts with all versions to locate a fault. We have examined all the cells in Table 10, and the accuracy improvement of all history versions is always more than that of half of all history versions.

It indicates that Delta4Ts can improve the effectiveness of fault localization techniques generally, and the more history version inputs, the higher improvement of accuracy Delta4Ts can produce.

4.5 Case study: debugging-in-parallel comparison

Another popular debugging manner is *debugging-in-parallel*. Different from Delta4Ts, such techniques focus on locating more than one fault at one shot. For example, Jones [40] proposed the first work in this direction. They showed that clustering could assist the fault localization on multi-fault programs. Hogerle et al. [35] is a recent such technique. Like the pioneer work in this family, it manages to cluster program runs to separate different faults, and outputs fault-localization results in each cluster to accelerate the process of debugging multi-fault programs. Abreu et al. [3] proposed a mode-based approach, which also located multiple faults at one shot. In their method, the number of faults needs to be given to initialize the model. This case study exhibits the difference between the debugging-in-parallel manner and Delta4Ts.

A two-fault buggy version from the project print_tokens is used in this case study. The first fault is on an assignment statement at line 280, and the second fault is on the condition statement of "if"-branch at line 313. We use "fault 1" and "fault 2" to refer to them.

Like previous section, we apply Delta4Ts to drive Tarantula to locate faults in it. First, Delta4Ts instrumented the faulty program to collect the execution information. Since the program did not have any historical versions in the first round, Delta4Ts just degenerated to the original Tarantula technique. The original Tarantula technique ranks the two faulty statements as 5 and 45, respectively. According to the result of Tarantula technique, we located "fault 1" with rank 5. Then, we fixed "fault 1" and reran Delta4Ts to locate the "fault 2" in the program. Again, Delta4Ts collected the execution information, made use of the fault-localization result of the original multi-fault program as signal, and generated a new sorted list. In the second round, we located "fault 2" with rank 11. Thus, the overall examining cost with respect to Delta4T is 5 and 11.

In comparison, we take the most representative technique [40] as example. The algorithm ended with three clusters. The ranks of the fault in these three clusters are {4, 65}, {7, 58}, and {307, 18}, respectively. [15] Here, {4, 65} denotes that the ranks of "fault 1"and "fault 2" are 4 and 65 in the ranked list obtained from the first cluster. The values of {7, 58} and {307, 18} are obtained from the second and

15. As the source code of the parallel debugging technique [40] is not available, we designed an algorithm which uses the clustering based on fault localization result for comparison and evaluation. First, we instrumented the buggy program to collect the execution information. Then, the test cases were clustered with KMeans algorithm [93] according to the program spectra from instrumentation. Next, we calculated the suspiciousness value with each cluster of test cases.

the third clusters, respectively. In other words, the least cost to locate the two faults are 4 and 18 accordingly.

From the case study, we have the following observations. First, when running with no history information given, Delta4Ts degenerates to a conventional fault-localization technique, and the debugging-in-parallel approach performs better. Second, with the aid of history information, Delta4Ts shows improvements in fault-localization effectiveness over the debugging-in-parallel approach. Third, since the two approaches are for different purposes, we conclude that both of them are effective to accelerate fault-localization.

4.6 Threats to validity

In this section, we discuss the threats to validity of our experiment.

4.6.1 Internal Validity

We follow the SIR documents to manipulate the common data set to simulate a software development scenario. Experiments in realistic scenario may manifest different observations. We use gcov to conduct the coverage profiling. The tool gcov is a good coverage profiling tool except handling the stack-overflow error. To give a better explanation for the crashing cases, we use some tactics associated with gcov to retrieve the traces for the exceptional cases. Previous work [21], [110] also investigated this topic, as exception information in runtime contained plenty of error information, thus providing good support for fault localization. In this paper, we take the runtime exception runs as failed runs. Different configurations on a coverage profiling tool may result in different observations. Further, to identify the history of a block, we adapt a historical slicing tool [81], [82] to help us complete this experiment. However, historical slicing techniques cannot identify the correct line equivalence in different revisions. This inherent limitation may negatively influence the results of our experiments.

4.6.2 Construct Validity

Construct validity refers to whether the experiment actually measures what it intends to measure.

In this experiment, we include Op2, Jaccard, Tarantula, Wong2, Wong1, Rogot1, D*, Ochiai, and Cohen for comparison. Also, there exist other statement-level techniques. Different statement-level techniques may have different results. There exist other techniques not included. For example, both CBI and SOBER are representative predicate-level fault-localization techniques and popularly used to compare with new predicate-level fault localization techniques.

We use the metrics Eff, Exp, and Sav to measure the effectiveness of a technique. These metrics are originally adapted from their statement-level fault localization counterparts. We realize that these metrics may give biased evaluation results in some extreme cases. We also extend the data analysis using the other metrics including Top-5, Top-5%, and Top-5‰. The use of other metrics may also produce different comparison results. Future investigations are necessary.

In the experiment, a lot of faulty program versions are generated. However, some faulty statements are not executable but have an influence on the program states in different versions. To compute the fault localization cost, we manually choose the statements close to them or some statements which have some kind of relationship with them. Such manual work may cause threats to construct validity. Different faulty versions and different faulty executable statements selections may also produce different results [86]. Although we excluded multi-fault programs, in which multiple faults did not coexist, we may make some mistakes that the experiment has many problematic programs. Moreover, fault localization modes may affect the conclusions. When evaluating the accuracy of SBFL in multi-fault programs, one fault is identified and fixed at a time in our experiment. Many faults are identified and fixed at a time, which probably affect the accuracy measures of SBFL. This is because software faults may have interactions between each other in various ways [87], and these interaction may influence the execution of test cases and further leads to different results.

The law of large numbers is used during the model development in this paper. However, no realistic experiment can provide enough number of versions, which is directly related to how we can rely on the conducted experiment to validate the technical correctness of our model.

There exist kinds of static impact that cannot be removed by our model. For example, "`return (x + z > y) ? (x) : (y)`" is a very effective syntactic sugar of C/C++, JAVA, and so on. It evaluates the value of an expression and accordingly takes one action from two candidates. Our experiences from these examples are that the suspiciousness of such a statement is the mix up of two parts, i.e., `x` and `y`. If one of them is fault-relevant, the whole statement is inflated as superposition of suspiciousness from a faulty entity and suspiciousness from a non-faulty one.

4.6.3 External Validity

One external validity concerning the effectiveness of our model is the representativeness of the subject programs. Although some of the subjects are real-life and large-scale programs, they only represent limited classes of programs. Using other programs in the experiment may produce different results.

Another external threat to validity is the programming characteristics. In our experiment, all subjects are writen in either C or Java. However, different programming characteristics result in different program structures and thus significantly influence the performance of Delta4Ts.

Moreover, there are passed test cases and failed test cases for each version in our experiment. Delta4Ts does not fit the situation that the basic SBFL cannot work because only failed test cases exist.

Program evolution includes not only minor modification on statements, but also code refactoring that may entirely change program structures . Algorithm 1 explains how we address basic blocks that cannot be tracked in some program versions. However, we also realize that the subjects selected in the experiment may not adequately evaluate all properties of our proposal. According to our model, we foresee that: (1) Basic blocks deleted from the new version will receive no evaluation by current fault-localization methods.

Faults related to them will be located by evaluating their directly affected or directly adjacent basic blocks. (2) Basic blocks inserted in the new version will receive no suspiciousness adjustment. Their suspiciousness scores are not affected by Delta4Ts. Other subjects having such cases may result in different experimental results and observations.

5 RELATED WORK

A typical debugging process is comprised of program comprehension, fault localization, fixing, and retesting. In the past decades, researchers have proposed many automated fault localization techniques to assist this time-consuming yet critical activity. We briefly list out related work of this paper in the following sections.

5.1 SBFL Techniques

The idea of referencing prior knowledge or learning knowledge to facilitate fault localization is not novel. For example, Liu et al. [56] used the information of fault classification to adjust code suspiciousness. Sohn and Yoo [84] made use of code and change metrics to improve fault localization. Neelofar et al. [62] designed fault-localization formula by learning parameters to drive hyperbolic functions. In this paper, we propose a technique, which improves fault-localization accuracy by referencing debugging history and learning to recognize false signals in code suspiciousness. In the following subsections, we list out related techniques in three main groups.

5.1.1 Statement-Level Techniques

In the family of SBFL techniques, many techniques and methods are designed for localizing the fault on the statement-level.

Tarantula, which is proposed by Jones et al. [39] and is considered to be the first SBFL technique, models likelihood of fault existence as the suspiciousness score. It employs the proportions of passed and failed test cases to assess the suspiciousness of each statement in a program. There are many variants of statement-level SBFL techniques. Baudry et al. [8] reported the observation that some groups of statements were always executed by the same set of test cases. Based on this observation, they presented an approach to maximize the number of dynamic basic blocks by identifying out a subset of original test set. Similarly, Wong et al. [100] proposed a coverage-based method to prioritize suspiciousness code. Besides the aforementioned methods, many statement-level SBFL techniques are proposed, including Jaccard [1], Ochiai [2], Wong2 [100] and so on. Essentially, these techniques designed different formulae to compute the suspiciousness of a statement and improve the accuracy. Different from these work, by leveraging the program structures, our work presents an approach that can be applied to any SBFL techniques to improve the performance.

5.1.2 Predicate-Based Techniques

Predicate-based fault localization techniques have also been explored in locating faults.

Liblit et al. [46] developed CBI, which used the number of times a predicate being evaluated true in passed and failed program runs with reconciling the specificity and sensitivity of the predicates to estimate the suspiciousness of predicates. However, for those predicates that are always evaluated true, CBI was ineffective. SOBER [49] compared the distributions of evaluation biases between passed runs and failed runs to obtain the suspiciousness of predicates. The larger the value of evaluation biases, the more likely the predicate associates with the root cause of failure. Chilimbi et al. [10] conducted SBFL using paths instead of predicates in their work Holmes. Yu et al. [109] utilized the combination of statements, predicates, and data dependencies to conduct fault localization.

We have elaborated on the principle of Delta4Ts to remove false signals and these false signals also appear in predicate-based fault localization techniques. Because Delta4Ts leverages the program structure but is not limited on the source code level, we believe that Delta4Ts is also applicable to predicate-level SBFL techniques. However, more experiments are needed to validate this idea. Other program entities have also been adopted as fault indicators.

5.1.3 Other Kinds of SBFL Techniques

To achieve the goal of fault localization, different kinds of approaches are carried out.

Mutation-based techniques (e.g., [55]) estimate location of faults by generating mutants at different program sites and monitoring the failing rate of the generated program versions. The basic logic is that when an additional fault is embedded into a faulty program, it has a high chance to increase the failing rate of the program under test. On the other hand, when a faulty statement is further altered, the change in test outcome (pass or fail) of program runs may not be huge. Moreover, to reduce the time-to-release of a program, Jones et al. [40] considered to locate faults in parallel. They used clustering technique to divide failed test cases into disjoint subsets, of which each was combined with passed test cases to assist in locating a particular fault. Abreu et. al [3] used dynamic information to model of the analyzed program and employed logic reasoning over program traces to improve the localization results. Abreu et. al [4] further proposed a framework to combine model-based diagnosis approach and spectrum-based fault localization, coined BARINEL. They extracted abstract information from program spectra to model one program and used Bayesian reasoning to deduce multiple-fault candidates. SBFL and slicing-hitting-set-computation (SHSC) came with small computational overhead and aid programmers to identify possible locations of faults more efficiently. However, SHSC resulted in an undesirable high ranking of statements which were executed in many test cases, such as constructors. SBFL operates on block level. Pill et al. [9] combined SHSC with SBFL in order to improve the ranking of faulty statements.

Our method can also be applied on them since Delta4Ts manipulates code suspiciousness, which is also available with such kind of techniques.

5.2 Empirical and Theoretical Analysis of the SBFL Formula

The formula of computing suspiciousness score is the kernel of an SBFL technique. Because various SBFL formulae

have been designed and proposed, some research groups conducted empirical studies to investigate the effectiveness of various SBFL techniques. Jones et al. [38] performed an empirical study to compare the effectiveness of Tarantula with four other techniques. They found Tarantula, on the same set of subjects, consistently outperformed the other four techniques regarding both effectiveness and efficiency. Similarly, Abreu et al. [2] evaluated the similarity coefficients of Tarantula, Jaccard, Ochiai, and AMPLE. They found Ochiai generated the best coefficient with these four techniques.

Further, on the other hand, researchers also theoretically analyzed the effectiveness of these fault-localization techniques and investigated the relationship between them. Lee et al. [45] make the first step for analyzing the relationship between SBFL formulae theoretically. They proved formulae Tarantula and q_e are theoretically equivalent. Naish et al. [60] conducted an extensive analysis to prove some SBFL formulae are algebraically equivalent to one another. Xie et al. [103] discovered a partial order relation over a set of 30 SBFL formulae to locate the fault in an arbitrary program containing a single fault. They proved that many SBFL formulae can be placed into six groups of formulae such that the formulae in each group share the same accuracy and two of the six groups contain formulae that no other formulae they analyzed can be more accurate than the former formulae. Yoo [108] further reported both theoretical evidence for the human competitiveness of the evolved fault localization formulæ under the single fault scenario. They also conducted empirical study to enhance their proof. On the other hand, recent studies by Pearson et al. [64] and Keller et al. [41], however, suggest that experimentation on artificial faults and test suites lead to spurious differences between SBFL techniques. They also claim real faults tend to be more complex and greatly decrease fault-localization performance. Xie et al. [102] demonstrated genetic programming can automatically design globally optimal formula.

In spite of pairs of formulae having known deterministic partial orders of accuracy comparison drawn from theoretical analysis, empirical studies are also necessary to compare the other pairs of formulae. Zhang et al. [117] conducted both analytical analysis and empirical evaluation to report such results. Pearson et al. [64] reported experimental results on programs containing real faults. Tang et al. [89] proposed an empirical framework to report the actual accuracy of any formula in controlled settings.

5.3 Impact Factors to SBFL

The accuracy of an SBFL formula to locate faults is influenced by a number of factors, such as the program, fault, test suite issues or program scale [41]. Thus, there is another extensively studied dimension that researchers develop techniques to address the program, fault or test suite issues which adversely affect the accuracy of SBFL formulae.

To eliminate or reduce the impacts on effectiveness of the factors, a number of techniques have been invented. *Coincidental correctness* refers to a program run that a fault along the program run has been activated but no failure is produced [75]. Wang et al. [95] proposed a pattern-based methodology to refine the code coverage achieved by a set of program runs on each program entity. Their technique is to match the data-flow and control-flow along each run with each given context pattern, which models how a specific kind of fault (e.g., missing assignment) at that program entity is triggered and propagated to the program output. Masri and Assi [58] showed that coincidental correctness is prevalent in both of its forms and demonstrated that it is a safety reducing factor for spectrum-based fault localization. They then used a clustering strategy to cleanse test suites from coincidental correctness to enhance the accuracy of SBFL techniques. *Background noise* has been reported as another factor that affects the accuracy of fault localization techniques. Liblit et al. [46] computed the probability of executing a predicate in the whole set of program runs to represent the *background noise*, and subtracted it from the probability of executing the same predicate in the set of failed program runs. In fact, many SBFL formulae also incorporate various variants of such a difference. Xu et al. [104] pointed out that a typical SBFL formula (denoted as A) is designed to exhibit a strong bias toward the set of failed program runs, and the suspiciousness score computed by the formula is only compared to a natural value (i.e., 0). They proposed to mirror a given SBFL formula (denoted as B) to exhibit the same bias (in the algebraic form) toward the set of passed program runs, and use the formula $A - B$ as the resultant formula for fault localization. *Partial evaluation* has been identified as yet another factor. For instance, Zhang et al. [112] reported that the *short-circuiting evaluations* behavior on predicates at the source code level affects the accuracy of predicate-based SBFL techniques. They empirically showed that differentiating the scenarios in the short-circuiting evaluation of the same predicate can improve the effectiveness of predicate-based SBFL techniques.

The impact of test cases to the accuracy of fault localization can be also important. Delta debugging constructs pair of passing and failing inputs in which difference is minimal to isolate state transitions in failing runs for debugging [11], [27]. Zhang et al. [115] proposed a method to figure out failure-causing parameter interactions using combinatorial concepts. Their methods are based on adaptive black-box testing, in which test cases are generated based on outcomes of previous tests. Their follow-up work [118] focuses on unlabelled test cases, i.e., test cases whose output is not identified, and showed that newly labelled test cases can improve the effectiveness of fault localization. Perez et al. [69] proposed a metrics to evaluate the diagnostic ability of a test case in term of fault localization effectiveness. Besides, both Lu et al. [57] and us [116] tried to give fault-localization approaches when only the failed runs are available. The experimental results are promising, which can be explained as the abandon of unreliable data (passed runs may have coincidental correctness happened). *Class imbalance problem* has been reported. Zhang et al. [117] formulated the problem for SBFL research and presented a strategy that clones the set of failed test cases in a given test suite to make the expanded test suite as balanced as possible. They further mathematically proved that the test suite generated through their strategy can improve the accuracy of 19 existing SBFL formulae in the single-fault scenario than using the given test suite.

Different from above work, which concentrate on impact factors of input data (test case issues like [95]) or manipulation on input data (formula issues like [104]), the preliminary work of this paper [47] pointed out that program structure can be also an impact factor to localization precision. We argued that a better choice is to statistically capture the impact of the program structure on the effectiveness of fault localization and such an idea has been demonstrated to improve the effectiveness of Tarantula on the Siemens suite. In this paper, we systematically extended our previous work. Contrasting two versions to boost fault localization is not a new idea [70], [91], [111]. For example, Tang et al. [91] proposed a method DFL to compare the current version with the last history version to debug the current version. Our method is different from them in (i) our model is dedicatedly designed to address structure bias, (ii) we give a systematic analysis to the benefits of referencing history, and (iii) we propose the methodology to handle more than one history version.

Since Delta4Ts addresses an impact factor orthogonal to all above related work. An illustration to exclude coincidental correctness and apply MinusFKBC [104] on the examples in Fig 1 is as follows: By excluding all the coincidental test cases to drive Wong1 to locate the fault in the versions $\mathcal{V}^1$, $\mathcal{V}^2$, and $\mathcal{V}^3$ of the example, the values of $\mathrm{a_{ef}}$ of each block are not changed, and the values of $\mathrm{a_{ep}}$ of $\langle\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$, $\mathcal{B}_4\rangle$ are changed to $\langle 0, 5, 3, 2\rangle$ in $\mathcal{V}^1$, $\langle 7, 0, 0, 0\rangle$ in $\mathcal{V}^2$ and $\langle 0, 1, 0, 2\rangle$ in $\mathcal{V}^3$, respectively. We find that the test suite, the coverage spectra, and the suspicious values of each block have all been changed. However, the changes are not enough to affect the ranking order of the basic blocks, in all three versions of the example. Based on this step, we continue to apply MinusFKBC [104] to reduce noise and rerun Wong1, this time we find that the suspicious values of some blocks have also been changed in the three versions. Coincidentally, the ranking orders in no version have been affected. Based on the above two steps, we input the altered test cases to Delta4Ts to drive Wong1. We find that the rank of $\mathcal{B}_2$ in $\mathcal{V}^2$ is raised from 2 to 1, and the rank of $\mathcal{B}_3$ in $\mathcal{V}^3$ is raised from 3 to 1, which indicates that Delta4Ts can improve the effectiveness of Wong1 on versions $\mathcal{V}^2$ and $\mathcal{V}^3$. In summary, with altered test suite, coverage spectra, and the suspicious values of basic blocks, Delta4Ts can still improve the accuracy of fault localization on this example. The example demonstrates the possibility to apply multiple boosting strategies simultaneously. More studies on integrating different methods may disclose interesting findings further.

5.4 Other fault-localization approaches

Apart from the techniques reviewed above, there are many other kinds of fault localization studies.

The mathematical foundations of model-based diagnostics or diagnosis [14], [72] from first principles was laid by Reiter and Kleer, who proposed model-based diagnosis to diagnose system failures. Model-based diagnosis [14], [22], [32], [67], [68], [72] was to infer the behavior of the composite system (software) from knowledge of the structure and function of the individual components comprising the system (software). Diagnostic tasks required determining the differences between a model of an artifact and the artifact itself. The differences between the manifested behavior of the artifact and the predicted behavior of the model guided the search for the differences between the artifact and its model. Pill and Wotawa [67] focused on combining combinatorial testing with consistency-oriented model-based diagnosis.

Zhang et al. [113] proposed an approach, which captured the propagation of infected program states through the edges in a control flow graph. It associated scores of control flow edges to suspiciousness scores of basic blocks to locate faults. Zhang et al. [119] approached in a similar way. Their work were evaluated effective in boosting fault localization accuracy. Gopinath et al. [26] combined specification-based analysis with dynamic test execution information to produce effective localization consistently across varying fault and specification complexity.

Some methods about test case are also used to improve fault localization. Gong et al. [24] designed a diversity-maximization-speedup approach to reduce the manual labeling of test cases and improved the accuracy of fault localization. Yoo et al. [107] addressed the problem of fault localization prioritization to maximize the fault localization by ranking remaining test cases once a previous fault is found. Xuan et al. [105] proposed a novel concept of spectrum driven test case purification for improving fault localization

Approaches based on information retrieval are also used to localize faults [52], [79]. Nguyen et al. [52] proposed a model based program features and trained it with historical data to reduce triaging efforts and save time for developers in fixing bugs. Saha et al. [79] take advantage of structured information retrieval based on code constructs to enable more accurate bug localization.

In the last few decades, software systems become more complicated than ever and many programs are likely to be written in more than one programming language. Fault localization is also conducted for systems involving more than one programming language or programming language for dedicated use (e.g., [30], [54]). Hong et al. [33] propose a mutation-based fault localization technique for real-world multilingual programs. The method proposed in this paper is highly expected to be integrated with these above new directions of the fault-localization research field.

5.5 Other comprehensive studies

Most existing techniques focus on generating candidate set from potentially faulty statements and ranking them according to their assessed values. Parnin and Orso [65] pointed out that most of work ignored how to understand the root cause of a failure, which typically involved complex activities. They investigated how developers used and benefited from automated debugging tools through a set of human studies. They observed that sometimes developers did not correct this fault, but turned to modify non-fault code to bypass the failure by using conventional debugging, and pointed out that an automated debugging tool may help ensure developers correct faults instead of simply patching failures. Moreover, they observed that even if the tool did not pinpoint the precise location of the fault (say, displaying relevant code entrance points), it could help program understand the problem considerably.

They further suggested that (1) techniques should focus on improving absolute rank rather than percentage rank; (2) debugging tools can be more successful if they focused on searching through or automatically highlighting certain suspicious statements; (3) research should focus on providing an ecosystem that supports the entire tool chain for fault localization, including managing and orchestrating test cases, and (4) researchers should perform more human studies to understand how the use of richer information (e.g., slices, test cases, values) can make debugging aids more useable. Ang et al. [7] further investigated the progress in the four aspects. They found that the SBFL research community was adopting the absolute evaluation metric [44], [71], [98]. Our work is partially inspired in a way very similar to their work. Our motivation is that in practice some statements are always ranked high and the programmer can remember them and skip them each time. By recognizing them as bias data, Delta4Ts effectively improves the effectiveness of fault localization.

Furthermore, researchers have proposed several techniques to improve result comprehension [29], [48], [66], [96], [97], [120]. Unfortunately, substantially less effort has been put in developing ecosystems [29], [66] and performing user studies [29], [42], [94], which play essential roles in closing the gap between research and practice. They suggested the SBFL research community to focus on creating an ecosystem that can be used by developers during debugging activities.

Another thread is the impact of fault localization on program repair. SBFL techniques, which produce a rank list of statements in descending order of their suspiciousness values, are nowadays widely used in current automated program repair studies [53], [59], [106]. For example, Nguyen et al. [53] examined one buggy statement at a time from a ranked list of statements reported by SBFL, and further derived and solved repair constraints to fix bugs. Apparently, the higher the rank of fault statements in ranked list, the less the cost of the similar method to repair the faults is. We also want to transfer the idea of Delta4Ts to program repair by suppressing *fake* repair candidates appeared in the history. However, how to make use of history information to improve program repair is still unknown.

6 Conclusion

Existing SBFL techniques utilize the executing information to estimate the positions of faults and narrow down the region of the faulty statements. However, most of them pay little attention to the biases caused by static program characteristics like program structure (including programming patterns).

In this paper, we have studied the impacts of such structural characteristics of a program on the accuracy of SBFL techniques, formally formulate the problem of structure bias, and put forward a theoretical model to capture and estimate the impacts of structure bias on an SBFL technique and remove them. We have conducted a controlled experiment on nine representative techniques over twelve subjects with thousands of multi-fault versions to evaluate Delta4Ts. The experimental result has confirmed the existence of the concerned impacts and validated the usefulness of Delta4Ts in improving the accuracy of nine SBFL techniques Op2, Jaccard, Tarantula, Wong2, Wong1, Rogot1, Ochiai, D*, and Cohen. Empirical results have shown that using Delta4Ts, the accuracy of SBFL techniques can be improved by 34.78% on 12 C programs and 30.6% on 6 Java programs, in the experiment.

We also realize that the improvements over conventional SBFL is not a surprise, since extra information are used. In other words, the worst case is that when no history information is given, Delta4Ts always degenerates to a conventional SBFL technology. On the other hand, the case study suggests that a promising future direction can be the integration of Delta4Ts with the debugging-in-parallel approach. Furthermore, though Delta4Ts successfully shows its effectiveness in eliminating impacts from structure bias, how to handle the structure bias in a comprehensible way is still unknown. Other future research includes a thorough study on other fault localization techniques and evaluation on other realistic subject programs.

Appendix

Raw results (Table 11)

Acknowledgment

This work was supported by a grant from the National Key Basic Research Program of China (project no. 2014CB340702), a grant from the National Natural Science Foundation of China (project no. 61379045), a grant from the Key Research Program of Frontier Sciences, Chinese Academy of Sciences (project no. QYZDJ-SSW-JSC036), a grant from the China Scholarship Council (project no. 201604910232), and the GRF of HKSAR Research Grants Council (project nos. 11214116 and 11200015), the HKSAR ITF (project no. ITS/378/18), the CityU MF_EXT (project no. 9678180), the CityU SRG (project nos. 7004882 and 7005216).

References

[1] R. Abreu, P. Zoeteweij, and A.J.C. van Gemud. (2007). “On the accuracy of spectrum-based fault localization”. In *Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques - Mutation (TAICPART-MUTATION 2007)*, pages 89–98.

[2] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. (2006). “An evaluation of similarity coefficients for software fault localization”. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC 2006)*, pages 39–46.

[3] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. (2008). “An observation-based model for fault localization”. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 64–70.

[4] R. Abreu, P. Zoeteweij, and A.J.C. van Gemund. (2009). “Spectrum-based multiple fault localization”. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE 2009)*, pages 88–99.

[5] A. Alali, H. Kagdi, and J.I. Maletic. (2008). “What's a typical commit? a characterization of open source software repositories”. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC 2008)*, pages 182–191.

[6] M.R. Anderberg. “Cluster Analysis for Applications”. *Probability and Mathematical Statistics New York Academic Press (1973)*, pages 347–353.

[7] A. Ang, A. Perez, A. van Deursen, Arie and R. Abreu. (2017). “Revisiting the practical use of automated software fault localization techniques”. In *Proceedings of 2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW 2017)*, pages 175–182.

TABLE 11: Accuracy Change (in Eff) when applying Delta4Ts to drive a technique to locate fault.

Tests on 5-fault programs

Op2 on print_tokens: 77 → 77, 86 → 86, 96 → 17, 95 → 11, 30 → 3.
Jaccard on print_tokens: 3 → 3, 3 → 2, 7 → 7, 22 → 12, 30 → 3.
Tarantula on print_tokens: 3 → 3, 3 → 2, 7 → 7, 22 → 12, 30 → 3.
Wong2 on print_tokens: 12 → 12, 12 → 12, 8 → 8, 62 → 18, 66 → 16.
Wong1 on print_tokens: 77 → 77, 86 → 6, 96 → 17, 95 → 11, 44 → 23.
Rogot1 on print_tokens: 3 → 3, 3 → 2, 7 → 7, 55 → 24, 55 → 15.
Ochiai on print_tokens: 3 → 3, 3 → 1, 7 → 7, 14 → 4, 30 → 3.
Cohen on print_tokens: 3 → 3, 3 → 2, 7 → 7, 38 → 13, 30 → 3.
D* on print_tokens: 3 → 3, 3 → 2, 7 → 7, 22 → 12, 30 → 3.
Op2 on print_token2: 60 → 60, 53 → 19, 73 → 15, 55 → 11, 33 → 4.
Jaccard on print_token2: 2 → 2, 25 → 17, 52 → 35, 67 → 35, 58 → 46.
Tarantula on print_token2: 2 → 2, 25 → 16, 52 → 33, 67 → 34, 58 → 47.
Wong2 on print_token2: 11 → 11, 26 → 9, 54 → 26, 79 → 37, 102 → 46.
Wong1 on print_token2: 78 → 78, 75 → 25, 74 → 28, 63 → 19, 54 → 23.
Rogot1 on print_token2: 2 → 2, 26 → 14, 54 → 34, 73 → 48, 71 → 50.
Ochiai on print_token2: 2 → 2, 28 → 8, 58 → 22, 77 → 36, 68 → 30.
Cohen on print_token2: 2 → 2, 24 → 17, 54 → 34, 69 → 49, 55 → 44.
D* on print_token2: 2 → 2, 25 → 17, 55 → 36, 77 → 40, 57 → 47.
Op2 on replace: 46 → 46, 41 → 21, 40 → 19, 36 → 17, 23 → 18.
Jaccard on replace: 9 → 9, 24 → 19, 29 → 23, 36 → 28, 40 → 31.
Tarantula on replace: 9 → 9, 24 → 19, 29 → 23, 36 → 29, 40 → 32.
Wong2 on replace: 21 → 21, 37 → 25, 51 → 33, 62 → 36, 71 → 51.
Wong1 on replace: 50 → 50, 48 → 26, 50 → 27, 50 → 27, 49 → 41.
Rogot1 on replace: 13 → 13, 30 → 21, 45 → 28, 57 → 35, 68 → 51.
Ochiai on replace: 8 → 8, 24 → 14, 28 → 18, 30 → 22, 30 → 22.
Cohen on replace: 10 → 10, 25 → 20, 31 → 25, 36 → 29, 42 → 32.
D* on replace: 9 → 9, 24 → 19, 29 → 24, 36 → 28, 41 → 32.
Op2 on schedule: 28 → 28, 28 → 14, 21 → 11, 8 → 5, 3 → 3.
Jaccard on schedule: 3 → 3, 4 → 4, 5 → 5, 5 → 4, 3 → 2.
Tarantula on schedule: 3 → 3, 4 → 4, 5 → 5, 5 → 4, 3 → 2.
Wong2 on schedule: 6 → 6, 6 → 5, 5 → 4, 8 → 6, 12 → 12.
Wong1 on schedule: 34 → 34, 34 → 16, 33 → 20, 28 → 17, 36 → 19.
Rogot1 on schedule: 3 → 3, 4 → 3, 4 → 4, 7 → 6, 11 → 11.
Ochiai on schedule: 3 → 3, 5 → 3, 5 → 5, 5 → 4, 3 → 2.
Cohen on schedule: 3 → 3, 4 → 3, 4 → 4, 5 → 4, 3 → 3.
D* on schedule: 3 → 3, 4 → 4, 5 → 5, 5 → 4, 3 → 2.
Op2 on schedule2: 10 → 10, 19 → 16, 27 → 23, 39 → 31, 47 → 32.
Jaccard on schedule2: 40 → 40, 59 → 55, 63 → 54, 61 → 43, 62 → 28.
Tarantula on schedule2: 41 → 41, 60 → 45, 63 → 39, 62 → 45, 63 → 41.
Wong2 on schedule2: 38 → 38, 45 → 30, 46 → 17, 46 → 25, 46 → 32.
Wong1 on schedule2: 34 → 34, 35 → 34, 37 → 35, 43 → 38, 47 → 38.
Rogot1 on schedule2: 38 → 38, 48 → 45, 47 → 44, 47 → 34, 48 → 27.
Ochiai on schedule2: 39 → 39, 59 → 44, 63 → 38, 62 → 46, 63 → 46.
Cohen on schedule2: 41 → 41, 60 → 55, 63 → 53, 62 → 42, 63 → 30.
D* on schedule2: 40 → 40, 59 → 55, 63 → 54, 61 → 42, 62 → 28.
Op2 on tcas: 9 → 9, 11 → 8, 14 → 9, 24 → 17, 30 → 20.
Jaccard on tcas: 8 → 8, 12 → 10, 16 → 12, 25 → 18, 32 → 20.
Tarantula on tcas: 8 → 8, 12 → 9, 16 → 12, 25 → 18, 32 → 20.
Wong2 on tcas: 8 → 8, 12 → 10, 17 → 13, 24 → 17, 28 → 21.
Wong1 on tcas: 22 → 22, 23 → 18, 23 → 15, 25 → 17, 27 → 19.
Rogot1 on tcas: 8 → 8, 12 → 10, 18 → 14, 25 → 19, 28 → 20.
Ochiai on tcas: 8 → 8, 12 → 8, 16 → 10, 26 → 18, 33 → 24.
Cohen on tcas: 8 → 8, 12 → 10, 16 → 12, 25 → 18, 32 → 20.
D* on tcas: 8 → 8, 12 → 10, 16 → 12, 25 → 18, 32 → 20.
Op2 on tot_info: 3 → 3, 4 → 4, 6 → 5, 8 → 7, 17 → 15.
Jaccard on tot_info: 7 → 7, 22 → 19, 31 → 23, 36 → 31, 42 → 35.
Tarantula on tot_info: 42 → 35, 7 → 7, 22 → 19, 31 → 23, 36 → 31.
Wong2 on tot_info: 24 → 24, 47 → 13, 56 → 16, 65 → 29, 71 → 48.
Wong1 on tot_info: 7 → 7, 8 → 7, 11 → 9, 16 → 15, 23 → 23.
Rogot1 on tot_info: 24 → 24, 46 → 23, 53 → 27, 64 → 34, 71 → 43.
Ochiai on tot_info: 14 → 14, 31 → 16, 37 → 21, 38 → 27, 41 → 32.
Cohen on tot_info: 14 → 14, 35 → 27, 43 → 33, 44 → 35, 46 → 35.
D* on tot_info: 7 → 7, 22 → 19, 31 → 24, 36 → 31, 42 → 36.
Op2 on flex: 353 → 353, 409 → 409, 618 → 618, 905 → 817, 821 → 545.
Jaccard on flex: 460 → 460, 650 → 650, 989 → 989, 1379 → 1298, 1124 → 877.
Tarantula on flex: 460 → 460, 650 → 650, 989 → 989, 1379 → 1298, 1124 → 877.
Wong2 on flex: 444 → 444, 630 → 630, 885 → 885, 1180 → 1092, 1171 → 895.
Wong1 on flex: 762 → 762, 762 → 762, 766 → 766, 771 → 713, 790 → 451.
Rogot1 on flex: 444 → 444, 625 → 625, 881 → 881, 1180 → 1095, 1171 → 923.
Ochiai on flex: 414 → 414, 573 → 573, 942 → 942, 1381 → 1293, 1126 → 850.
Cohen on flex: 460 → 460, 650 → 650, 989 → 989, 1379 → 1298, 1124 → 877.
D* on flex: 460 → 460, 650 → 650, 989 → 989, 1379 → 1298, 1124 → 877.
Op2 on grep: 26 → 26, 105 → 106, 105 → 48, 115 → 58, 136 → 136.
Jaccard on grep: 26 → 26, 136 → 136, 45 → 44, 40 → 37, 891 → 298.
Tarantula on grep: 26 → 26, 136 → 136, 45 → 44, 40 → 37, 863 → 298.
Wong2 on grep: 26 → 26, 219 → 97, 46 → 41, 41 → 40, 422 → 403.
Wong1 on grep: 42 → 42, 263 → 178, 267 → 43, 274 → 58, 314 → 314.
Rogot1 on grep: 26 → 26, 138 → 138, 45 → 44, 40 → 37, 386 → 386.
Ochiai on grep: 26 → 26, 130 → 130, 105 → 43, 115 → 27, 145 → 145.
Cohen on grep: 26 → 26, 138 → 138, 45 → 44, 40 → 37, 299 → 299.
D* on grep: 26 → 26, 136 → 136, 45 → 44, 40 → 37, 298 → 298.
Op2 on gzip: 350 → 350, 341 → 283, 346 → 7, 278 → 33, 117 → 7.
Jaccard on gzip: 318 → 24, 398 → 312, 374 → 362, 375 → 360, 361 → 24.
Tarantula on gzip: 318 → 318, 398 → 291, 374 → 362, 375 → 360, 361 → 24.
Wong2 on gzip: 317 → 317, 360 → 78, 350 → 7, 402 → 12, 14 → 5.
Wong1 on gzip: 217 → 217, 208 → 200, 221 → 133, 177 → 145, 117 → 62.
Rogot1 on gzip: 318 → 318, 388 → 291, 367 → 357, 402 → 360, 96 → 24.
Ochiai on gzip: , 414 → 414, 335 → 26, 417 → 7, 354 → 8.
Cohen on gzip: 318 → 318, 388 → 291, 361 → 355, 385 → 360, 361 → 24.
D* on gzip: 318 → 318, 398 → 314, 374 → 362, 375 → 360, 361 → 24.
Op2 on sed: 297 → 297, 427 → 165, 431 → 117, 286 → 72, 68 → 46.
Jaccard on sed: 11 → 11, 17 → 13, 29 → 27, 38 → 36, 82 → 47.
Tarantula on sed: 11 → 11, 21 → 14, 29 → 27, 38 → 35, 70 → 48.
Wong2 on sed: 6 → 6, 15 → 15, 34 → 19, 33 → 30, 144 → 109.
Wong1 on sed: 374 → 374, 445 → 179, 454 → 118, 368 → 73, 238 → 72.
Rogot1 on sed: 5 → 5, 19 → 12, 27 → 28, 36 → 36, 109 → 86.
Ochiai on sed: 9 → 9, 40 → 24, 28 → 39, 41 → 39, 41 → 41.
Cohen on sed: 5 → 5, 14 → 10, 26 → 30, 39 → 41, 70 → 48.
D* on sed: 12 → 12, 17 → 12, 29 → 29, 38 → 37, 87 → 48.
Op2 on space: 450 → 450, 717 → 237, 952 → 217, 1079 → 203, 369 → 236.
Jaccard on space: 98 → 98, 82 → 73, 93 → 71, 169 → 119, 220 → 220.
Tarantula on space: 98 → 98, 82 → 39, 93 → 37, 169 → 117, 220 → 219.
Wong2 on space: 75 → 75, 67 → 71, 76 → 76, 181 → 162, 539 → 506.
Wong1 on space: 607 → 607, 799 → 241, 998 → 231, 1070 → 199, 768 → 370.
Rogot1 on space: 85 → 85, 46 → 39, 49 → 40, 186 → 135, 497 → 410.
Ochiai on space: 82 → 82, 99 → 74, 95 → 85, 246 → 223, 184 → 140.

(to be continued)

(continue from previous page)

Cohen on space: 81 → 81, 39 → 37, 69 → 75, 197 → 169, 244 → 206.
D* on space: 98 → 98, 82 → 75, 93 → 72, 169 → 120, 220 → 219.
Ochiai on gzip: 414 → 414, 335 → 26, 417 → 7, 354 → 8, 26 → 7.
Cohen on gzip: 318 → 318, 388 → 291, 361 → 362, 385 → 360, 361 → 24.
D* on gzip: 318 → 318, 398 → 314, 374 → 362, 375 → 360, 361 → 24.
Op2 on sed: 297 → 297, 427 → 165, 431 → 117, 286 → 72, 68 → 46.
Jaccard on sed: 11 → 11, 17 → 13, 29 → 27, 38 → 36, 82 → 47.
Tarantula on sed: 11 → 11, 21 → 14, 29 → 27, 38 → 35, 70 → 48.
Wong2 on sed: 6 → 6, 15 → 15, 34 → 19, 33 → 30, 144 → 109.
Wong1 on sed: 374 → 374, 445 → 179, 454 → 118, 368 → 73, 238 → 72.
Rogot1 on sed: 5 → 5, 19 → 12, 27 → 28, 36 → 36, 109 → 86.
Ochiai on sed: 9 → 9, 40 → 24, 28 → 39, 41 → 39, 41 → 41.
Cohen on sed: 5 → 5, 14 → 10, 26 → 30, 39 → 41, 70 → 48.
D* on sed: 12 → 12, 17 → 12, 29 → 29, 38 → 37, 87 → 48.
Op2 on space: 450 → 450, 717 → 237, 952 → 217, 1079 → 203, 369 → 236.
Jaccard on space: 98 → 98, 82 → 73, 93 → 71, 169 → 119, 220 → 220.
Tarantula on space: 98 → 98, 82 → 39, 93 → 37, 169 → 117, 220 → 219.
Wong2 on space: 75 → 75, 67 → 71, 76 → 76, 181 → 162, 539 → 506.
Wong1 on space: 607 → 607, 799 → 241, 998 → 231, 1070 → 199, 768 → 370.
Rogot1 on space: 85 → 85, 46 → 39, 49 → 40, 186 → 135, 497 → 410.
Ochiai on space: 82 → 82, 99 → 74, 95 → 85, 246 → 223, 184 → 140.
Cohen on space: 81 → 81, 39 → 37, 69 → 75, 197 → 169, 244 → 206.
D* on space: 98 → 98, 82 → 75, 93 → 72, 169 → 120, 220 → 219.

Tests on 4-fault programs

Op2 on print_tokens: 83 → 83, 83 → 66, 58 → 12, 46 → 3.
Jaccard on print_tokens: 5 → 5, 19 → 10, 15 → 22, 94 → 3.
Tarantula on print_tokens: 5 → 5, 19 → 9, 15 → 15, 57 → 3.
Wong2 on print_tokens: 20 → 20, 38 → 38, 60 → 36, 86 → 56.
Wong1 on print_tokens: 83 → 83, 83 → 66, 69 → 33, 46 → 23.
Rogot1 on print_tokens: 5 → 5, 26 → 7, 55 → 48, 129 → 46.
Ochiai on print_tokens: 8 → 8, 4 → 3, 5 → 4, 3 → 3.
Cohen on print_tokens: 5 → 5, 23 → 12, 23 → 15, 101 → 5.
D* on print_tokens: 5 → 5, 19 → 11, 15 → 15, 110 → 3.
Op2 on print_token2: 60 → 60, 75 → 18, 89 → 10, 82 → 5.
Jaccard on print_token2: 40 → 40, 46 → 23, 83 → 39, 107 → 42.
Tarantula on print_token2: 40 → 40, 40 → 22, 80 → 39, 105 → 42.
Wong2 on print_token2: 46 → 46, 46 → 16, 130 → 32, 144 → 40.
Wong1 on print_token2: 78 → 78, 114 → 27, 131 → 25, 90 → 25.
Rogot1 on print_token2: 46 → 46, 73 → 22, 87 → 44, 105 → 44.
Ochiai on print_token2: 40 → 40, 116 → 12, 88 → 31, 86 → 33.
Cohen on print_token2: 40 → 40, 46 → 23, 79 → 44, 110 → 42.
D* on print_token2: 40 → 40, 46 → 24, 83 → 42, 112 → 42.
Op2 on replace: 55 → 55, 47 → 6, 38 → 10, 20 → 7.
Jaccard on replace: 12 → 12, 23 → 11, 27 → 11, 33 → 12.
Tarantula on replace: 12 → 12, 23 → 10, 35 → 9, 40 → 13.
Wong2 on replace: 96 → 96, 109 → 41, 112 → 43, 103 → 53.
Wong1 on replace: 57 → 57, 58 → 7, 56 → 12, 49 → 15.
Rogot1 on replace: 27 → 27, 52 → 44, 72 → 44, 102 → 51.
Ochiai on replace: 9 → 9, 26 → 19, 26 → 13, 26 → 19.
Cohen on replace: 13 → 13, 28 → 17, 38 → 21, 45 → 32.
D* on replace: 12 → 12, 23 → 10, 35 → 12, 40 → 13.
Op2 on schedule: 25 → 25, 24 → 13, 21 → 8, 3 → 2.
Jaccard on schedule: 4 → 4, 4 → 4, 5 → 4, 3 → 2.
Tarantula on schedule: 4 → 4, 4 → 4, 5 → 4, 3 → 2.
Wong2 on schedule: 5 → 5, 5 → 4, 7 → 6, 12 → 10.
Wong1 on schedule: 33 → 33, 33 → 15, 37 → 17, 36 → 17.
Rogot1 on schedule: 4 → 4, 4 → 3, 6 → 5, 11 → 11.
Ochiai on schedule: 4 → 4, 4 → 4, 5 → 5, 3 → 2.
Cohen on schedule: 4 → 4, 4 → 4, 5 → 4, 3 → 2.
D* on schedule: 4 → 4, 4 → 4, 5 → 4, 3 → 2.
Op2 on schedule2: 14 → 14, 28 → 26, 38 → 38, 50 → 43.
Jaccard on schedule2: 44 → 44, 62 → 56, 64 → 52, 66 → 36.
Tarantula on schedule2: 44 → 44, 62 → 46, 64 → 45, 66 → 45.
Wong2 on schedule2: 44 → 44, 53 → 31, 55 → 23, 57 → 35.
Wong1 on schedule2: 35 → 35, 41 → 36, 52 → 41, 56 → 47.
Rogot1 on schedule2: 44 → 44, 53 → 51, 56 → 42, 58 → 33.
Ochiai on schedule2: 43 → 43, 62 → 45, 65 → 45, 66 → 49.
Cohen on schedule2: 44 → 44, 62 → 55, 64 → 51, 66 → 37.
D* on schedule2: 44 → 44, 62 → 56, 64 → 52, 66 → 35.
Op2 on tcas: 10 → 10, 14 → 10, 20 → 15, 27 → 20.
Jaccard on tcas: 9 → 9, 16 → 13, 23 → 18, 29 → 19.
Tarantula on tcas: 9 → 9, 16 → 13, 23 → 18, 29 → 19.
Wong2 on tcas: 10 → 10, 15 → 11, 22 → 17, 28 → 23.
Wong1 on tcas: 23 → 23, 23 → 20, 25 → 20, 25 → 23.
Rogot1 on tcas: 10 → 10, 16 → 13, 23 → 19, 28 → 22.
Ochiai on tcas: 9 → 9, 16 → 13, 23 → 18, 30 → 19.
Cohen on tcas: 9 → 9, 15 → 13, 22 → 18, 29 → 18.
D* on tcas: 9 → 9, 16 → 5, 23 → 7, 29 → 14.
Op2 on tot_info: 5 → 5, 25 → 6, 30 → 9, 35 → 16.
Jaccard on tot_info: 10 → 10, 29 → 24, 37 → 30, 41 → 35.
Tarantula on tot_info: 30 → 30, 29 → 20, 37 → 26, 41 → 35.
Wong2 on tot_info: 30 → 30, 56 → 9, 68 → 14, 70 → 23.
Wong1 on tot_info: 30 → 30, 28 → 10, 34 → 26, 42 → 23.
Rogot1 on tot_info: 30 → 30, 55 → 20, 67 → 26, 70 → 30.
Ochiai on tot_info: 16 → 16, 34 → 31, 40 → 37, 39 → 37.
Cohen on tot_info: 17 → 17, 39 → 25, 46 → 30, 44 → 35.
D* on tot_info: 10 → 10, 29 → 26, 37 → 27, 41 → 30.
Op2 on flex: 438 → 438, 785 → 535, 1152 → 805, 969 → 874.
Jaccard on flex: 509 → 509, 785 → 786, 1209 → 1153, 1189 → 970.
Tarantula on flex: 509 → 509, 786 → 744, 1210 → 1018, 1189 → 993.
Wong2 on flex: 494 → 494, 744 → 770, 1081 → 739, 1238 → 532.
Wong1 on flex: 766 → 766, 770 → 739, 1020 → 783, 1016 → 794.
Rogot1 on flex: 493 → 493, 739 → 735, 1079 → 1033, 1238 → 950.
Ochiai on flex: 465 → 465, 735 → 725, 1196 → 1152, 1194 → 969.
Cohen on flex: 509 → 509, 785 → 785, 1209 → 1152, 1189 → 969.
D* on flex: 509 → 509, 785 → 95, 1209 → 77, 1189 → 171.
Op2 on grep: 42 → 42, 99 → 86, 135 → 33, 275 → 319.
Jaccard on grep: 48 → 48, 86 → 86, 34 → 33, 636 → 319.
Tarantula on grep: 48 → 48, 86 → 59, 34 → 34, 627 → 486.
Wong2 on grep: 41 → 41, 190 → 104, 147 → 44, 563 → 522.
Wong1 on grep: 86 → 86, 260 → 86, 266 → 38, 492 → 30.
Rogot1 on grep: 48 → 48, 86 → 94, 40 → 67, 540 → 208.
Ochiai on grep: 47 → 47, 94 → 86, 96 → 35, 350 → 328.
Cohen on grep: 48 → 48, 88 → 86, 37 → 33, 402 → 318.
D* on grep: 48 → 48, 150 → 86, 80 → 39, 401 → 42.
Op2 on gzip: 146 → 146, 257 → 236, 245 → 221, 292 → 231.
Jaccard on gzip: 74 → 74, 244 → 228, 257 → 245, 296 → 244.
Tarantula on gzip: 74 → 74, 244 → 71, 257 → 39, 296 → 42.
Wong2 on gzip: 132 → 132, 231 → 133, 238 → 128, 314 → 135.
Wong1 on gzip: 132 → 132, 177 → 228, 204 → 248, 169 → 289.
Rogot1 on gzip: 78 → 78, 240 → 126, 249 → 69, 314 → 42.

(to be continued)

(continue from previous page)

Ochiai on gzip: 78 → 78, 258 → 226, 282 → 243, 282 → 304.

Cohen on gzip: 74 → 74, 240 → 236, 249 → 246, 302 → 292.

D* on gzip: 286 → 286, 244 → 92, 257 → 84, 296 → 33.

Op2 on sed: 286 → 286, 372 → 19, 249 → 39, 46 → 41.

Jaccard on sed: 9 → 9, 27 → 20, 34 → 38, 50 → 41.

Tarantula on sed: 9 → 9, 27 → 24, 34 → 37, 49 → 74.

Wong2 on sed: 7 → 7, 116 → 28, 82 → 46, 112 → 100.

Wong1 on sed: 356 → 356, 424 → 19, 305 → 40, 202 → 57.

Rogot1 on sed: 10 → 10, 24 → 25, 47 → 44, 74 → 33.

Ochiai on sed: 10 → 10, 48 → 18, 36 → 37, 40 → 31.

Cohen on sed: 9 → 9, 30 → 19, 38 → 40, 45 → 41.

D* on sed: 99 → 99, 182 → 29, 193 → 34, 196 → 53.

Op2 on space: 103 → 103, 715 → 89, 846 → 84, 285 → 175.

Jaccard on space: 103 → 103, 91 → 60, 106 → 67, 182 → 177.

Tarantula on space: 103 → 103, 91 → 92, 106 → 131, 182 → 399.

Wong2 on space: 93 → 93, 213 → 124, 208 → 147, 445 → 236.

Wong1 on space: 651 → 651, 828 → 65, 917 → 112, 718 → 308.

Rogot1 on space: 82 → 82, 69 → 63, 129 → 113, 402 → 119.

Ochiai on space: 105 → 105, 72 → 52, 134 → 92, 154 → 166.

Cohen on space: 103 → 103, 57 → 90, 108 → 85, 203 → 177.

D* on space: 103 → 103, 91 → 90, 106 → 85, 182 → 177.

Tests on 3-fault programs

Op2 on print_tokens: 79 → 79, 46 → 23, 47 → 4.

Jaccard on print_tokens: 13 → 13, 17 → 10, 85 → 11.

Tarantula on print_tokens: 13 → 13, 17 → 10, 70 → 9.

Wong2 on print_tokens: 28 → 28, 64 → 45, 78 → 59.

Wong1 on print_tokens: 81 → 81, 59 → 34, 47 → 24.

Rogot1 on print_tokens: 16 → 16, 56 → 26, 114 → 50.

Ochiai on print_tokens: 7 → 7, 4 → 4, 5 → 4.

Cohen on print_tokens: 14 → 14, 24 → 13, 89 → 9.

D* on print_tokens: 13 → 13, 17 → 13, 90 → 9.

Op2 on print_token2: 67 → 67, 63 → 17, 27 → 18.

Jaccard on print_token2: 18 → 18, 50 → 33, 50 → 30.

Tarantula on print_token2: 18 → 18, 50 → 32, 49 → 31.

Wong2 on print_token2: 22 → 22, 60 → 27, 95 → 37.

Wong1 on print_token2: 83 → 83, 78 → 31, 61 → 22.

Rogot1 on print_token2: 20 → 20, 55 → 32, 67 → 35.

Ochiai on print_token2: 18 → 18, 56 → 25, 44 → 24.

Cohen on print_token2: 19 → 19, 51 → 34, 50 → 32.

D* on print_token2: 18 → 18, 50 → 35, 50 → 29.

Op2 on replace: 56 → 56, 40 → 5, 18 → 9.

Jaccard on replace: 15 → 15, 39 → 12, 57 → 13.

Tarantula on replace: 15 → 15, 38 → 12, 56 → 12.

Wong2 on replace: 86 → 86, 104 → 49, 110 → 60.

Wong1 on replace: 62 → 62, 55 → 10, 45 → 22.

Rogot1 on replace: 20 → 20, 75 → 42, 90 → 58.

Ochiai on replace: 14 → 14, 30 → 19, 39 → 31.

Cohen on replace: 15 → 15, 34 → 32, 58 → 46.

D* on replace: 15 → 15, 39 → 12, 53 → 16.

Op2 on schedule: 31 → 31, 33 → 17, 14 → 10.

Jaccard on schedule: 4 → 4, 5 → 5, 4 → 3.

Tarantula on schedule: 4 → 4, 5 → 5, 4 → 3.

Wong2 on schedule: 5 → 5, 8 → 6, 12 → 11.

Wong1 on schedule: 40 → 40, 43 → 22, 43 → 27.

Rogot1 on schedule: 4 → 4, 7 → 5, 11 → 10.

Ochiai on schedule: 4 → 4, 5 → 4, 5 → 4.

Cohen on schedule: 4 → 4, 5 → 5, 4 → 3.

D* on schedule: 4 → 4, 5 → 5, 4 → 3.

Op2 on schedule2: 21 → 21, 37 → 37, 53 → 46.

Jaccard on schedule2: 47 → 47, 66 → 58, 70 → 47.

Tarantula on schedule2: 47 → 47, 66 → 58, 70 → 47.

Wong2 on schedule2: 52 → 52, 64 → 34, 70 → 36.

Wong1 on schedule2: 37 → 37, 47 → 40, 46 → 43.

Rogot1 on schedule2: 52 → 52, 64 → 54, 71 → 42.

Ochiai on schedule2: 52 → 52, 64 → 54, 71 → 53.

Cohen on schedule2: 47 → 47, 66 → 58, 71 → 47.

D* on schedule2: 47 → 47, 66 → 58, 70 → 47.

Op2 on tcas: 14 → 14, 20 → 14, 23 → 18.

Jaccard on tcas: 11 → 11, 21 → 17, 27 → 17.

Tarantula on tcas: 11 → 11, 21 → 17, 27 → 17.

Wong2 on tcas: 12 → 12, 22 → 16, 28 → 24.

Wong1 on tcas: 23 → 23, 24 → 21, 24 → 21.

Rogot1 on tcas: 11 → 11, 22 → 18, 28 → 23.

Ochiai on tcas: 11 → 11, 21 → 15, 27 → 19.

Cohen on tcas: 11 → 11, 21 → 17, 27 → 17.

D* on tcas: 11 → 11, 21 → 17, 27 → 17.

Op2 on tot_info: 7 → 7, 8 → 7, 16 → 15.

Jaccard on tot_info: 18 → 18, 34 → 29, 38 → 33.

Tarantula on tot_info: 18 → 18, 34 → 29, 38 → 33.

Wong2 on tot_info: 39 → 39, 64 → 28, 68 → 41.

Wong1 on tot_info: 12 → 12, 15 → 14, 24 → 26.

Rogot1 on tot_info: 38 → 38, 63 → 34, 68 → 41.

Ochiai on tot_info: 22 → 22, 36 → 24, 36 → 28.

Cohen on tot_info: 25 → 25, 41 → 32, 40 → 33.

D* on tot_info: 18 → 18, 34 → 29, 38 → 33.

Op2 on flex: 494 → 494, 671 → 636, 842 → 675.

Jaccard on flex: 549 → 549, 919 → 890, 1064 → 918.

Tarantula on flex: 549 → 549, 919 → 890, 1064 → 918.

Wong2 on flex: 536 → 536, 868 → 833, 1099 → 929.

Wong1 on flex: 811 → 811, 837 → 803, 831 → 637.

Rogot1 on flex: 535 → 535, 865 → 835, 1099 → 951.

Ochiai on flex: 513 → 513, 884 → 850, 1060 → 892.

Cohen on flex: 549 → 549, 919 → 890, 1064 → 918.

D* on flex: 549 → 549, 919 → 890, 1064 → 919.

Op2 on grep: 64 → 64, 176 → 76, 225 → 123.

Jaccard on grep: 58 → 58, 79 → 78, 303 → 209.

Tarantula on grep: 58 → 58, 79 → 79, 302 → 209.

Wong2 on grep: 102 → 102, 252 → 141, 457 → 358.

Wong1 on grep: 159 → 159, 318 → 193, 412 → 365.

Rogot1 on grep: 45 → 45, 141 → 80, 351 → 296.

Ochiai on grep: 62 → 62, 113 → 104, 248 → 160.

Cohen on grep: 45 → 45, 100 → 92, 265 → 217.

D* on grep: 58 → 58, 79 → 77, 265 → 209.

Op2 on gzip: 146 → 146, 180 → 106, 169 → 57.

Jaccard on gzip: 91 → 91, 159 → 154, 208 → 205.

Tarantula on gzip: 91 → 91, 159 → 153, 208 → 205.

Wong2 on gzip: 86 → 86, 149 → 83, 219 → 56.

Wong1 on gzip: 135 → 135, 168 → 111, 148 → 115.

Rogot1 on gzip: 90 → 90, 155 → 153, 219 → 201.

Ochiai on gzip: 97 → 97, 176 → 100, 200 → 56.

Cohen on gzip: 90 → 90, 155 → 151, 212 → 205.

D* on gzip: 91 → 91, 159 → 155, 208 → 205.

Op2 on sed: 270 → 270, 259 → 62, 33 → 20.

Jaccard on sed: 13 → 13, 61 → 40, 33 → 23.

Tarantula on sed: 13 → 13, 59 → 40, 30 → 24.

Wong2 on sed: 14 → 14, 45 → 39, 53 → 41.

Wong1 on sed: 340 → 340, 328 → 82, 177 → 72.

Rogot1 on sed: 10 → 10, 37 → 32, 47 → 33.

Ochiai on sed: 16 → 16, 62 → 40, 26 → 19.

Cohen on sed: 14 → 14, 56 → 39, 30 → 24.

D* on sed: 13 → 13, 61 → 40, 33 → 23.

Op2 on space: 594 → 594, 627 → 181, 268 → 193.

Jaccard on space: 178 → 178, 91 → 66, 182 → 190.

Tarantula on space: 89 → 89, 91 → 36, 182 → 190.

Wong2 on space: 133 → 133, 113 → 97, 447 → 377.

Wong1 on space: 132 → 132, 795 → 226, 654 → 307.

Rogot1 on space: 994 → 994, 86 → 66, 400 → 333.

Ochiai on space: 211 → 211, 88 → 72, 173 → 143.

Cohen on space: 322 → 322, 76 → 64, 211 → 175.

D* on space: 742 → 742, 91 → 67, 182 → 192.

Tests on 2-fault programs

Op2 on print_tokens: 64 → 64, 39 → 6.

Jaccard on print_tokens: 14 → 14, 51 → 17.

Tarantula on print_tokens: 14 → 14, 51 → 17.

(to be continued)

(continue from previous page)

Wong2 on print_tokens:	43 → 43, 73 → 55.
Wong1 on print_tokens:	72 → 72, 38 → 41.
Rogot1 on print_tokens:	36 → 36, 63 → 41.
Ochiai on print_tokens:	6 → 6, 18 → 18.
Cohen on print_tokens:	19 → 19, 54 → 51.
D* on print_tokens:	14 → 14, 17 → 6.
Op2 on print_token2:	64 → 64, 5 → 5.
Jaccard on print_token2:	23 → 23, 36 → 29.
Tarantula on print_token2:	23 → 23, 36 → 29.
Wong2 on print_token2:	30 → 30, 59 → 37.
Wong1 on print_token2:	84 → 84, 55 → 22.
Rogot1 on print_token2:	26 → 26, 50 → 40.
Ochiai on print_token2:	22 → 22, 30 → 27.
Cohen on print_token2:	24 → 24, 37 → 30.
D* on print_token2:	23 → 23, 36 → 33.
Op2 on replace:	52 → 52, 17 → 4.
Jaccard on replace:	12 → 13, 38 → 13.
Tarantula on replace:	12 → 12, 38 → 13.
Wong2 on replace:	68 → 68, 66 → 65.
Wong1 on replace:	56 → 56, 44 → 11.
Rogot1 on replace:	20 → 20, 64 → 63.
Ochiai on replace:	10 → 10, 28 → 27.
Cohen on replace:	13 → 13, 40 → 35.
D* on replace:	12 → 12, 38 → 13.
Op2 on schedule:	30 → 30, 25 → 14.
Jaccard on schedule:	6 → 6, 6 → 4.
Tarantula on schedule:	6 → 6, 6 → 4.
Wong2 on schedule:	9 → 9, 12 → 11.
Wong1 on schedule:	44 → 44, 47 → 34.
Rogot1 on schedule:	8 → 8, 11 → 11.
Ochiai on schedule:	5 → 5, 7 → 5.
Cohen on schedule:	6 → 6, 5 → 4.
D* on schedule:	6 → 6, 6 → 4.
Op2 on schedule2:	34 → 34, 57 → 51.
Jaccard on schedule2:	52 → 52, 77 → 61.
Tarantula on schedule2:	52 → 52, 77 → 61.
Wong2 on schedule2:	63 → 63, 80 → 44.
Wong1 on schedule2:	40 → 40, 56 → 45.
Rogot1 on schedule2:	63 → 63, 80 → 59.
Ochiai on schedule2:	53 → 53, 77 → 60.
Cohen on schedule2:	53 → 53, 77 → 61.
D* on schedule2:	52 → 52, 77 → 61.
Op2 on tcas:	17 → 17, 22 → 18.
Jaccard on tcas:	16 → 16, 26 → 18.
Tarantula on tcas:	16 → 16, 26 → 18.
Wong2 on tcas:	18 → 18, 27 → 15.
Wong1 on tcas:	25 → 25, 24 → 14.
Rogot1 on tcas:	18 → 18, 28 → 17.
Ochiai on tcas:	16 → 16, 26 → 19.
Cohen on tcas:	16 → 16, 26 → 18.
D* on tcas:	16 → 16, 26 → 19.
Op2 on tot_info:	10 → 10, 15 → 14.
Jaccard on tot_info:	23 → 23, 34 → 30.
Tarantula on tot_info:	23 → 23, 34 → 30.
Wong2 on tot_info:	49 → 49, 62 → 38.
Wong1 on tot_info:	17 → 17, 25 → 24.
Rogot1 on tot_info:	48 → 48, 62 → 38.
Ochiai on tot_info:	24 → 24, 32 → 25.
Cohen on tot_info:	27 → 27, 36 → 30.
D* on tot_info:	23 → 23, 34 → 30.
Op2 on flex:	690 → 690, 873 → 784.
Jaccard on flex:	737 → 737, 1107 → 1036.
Tarantula on flex:	737 → 737, 1107 → 1036.
Wong2 on flex:	721 → 721, 1143 → 1053.
Wong1 on flex:	900 → 900, 866 → 734.
Rogot1 on flex:	720 → 720, 1143 → 1071.
Ochiai on flex:	722 → 722, 1107 → 1018.
Cohen on flex:	737 → 737, 1107 → 1036.
D* on flex:	737 → 737, 1107 → 1037.
Op2 on grep:	118 → 118, 183 → 97.
Jaccard on grep:	70 → 70, 196 → 180.
Tarantula on grep:	70 → 70, 196 → 181.
Wong2 on grep:	177 → 177, 412 → 301.
Wong1 on grep:	221 → 221, 350 → 297.
Rogot1 on grep:	111 → 111, 290 → 207.
Ochiai on grep:	82 → 82, 172 → 144.
Cohen on grep:	80 → 80, 206 → 194.
D* on grep:	70 → 70, 197 → 180.
Op2 on gzip:	161 → 161, 142 → 61.
Jaccard on gzip:	130 → 130, 171 → 168.
Tarantula on gzip:	130 → 130, 171 → 168.
Wong2 on gzip:	123 → 123, 179 → 62.
Wong1 on gzip:	160 → 160, 137 → 113.
Rogot1 on gzip:	128 → 128, 179 → 168.
Ochiai on gzip:	143 → 143, 165 → 61.
Cohen on gzip:	128 → 128, 174 → 168.
D* on gzip:	130 → 130, 171 → 169.
Op2 on sed:	237 → 237, 17 → 17.
Jaccard on sed:	46 → 46, 19 → 18.
Tarantula on sed:	46 → 46, 19 → 20.
Wong2 on sed:	44 → 44, 39 → 37.
Wong1 on sed:	330 → 330, 166 → 86.
Rogot1 on sed:	38 → 38, 26 → 25.
Ochiai on sed:	49 → 49, 17 → 17.
Cohen on sed:	48 → 48, 19 → 18.
D* on sed:	46 → 46, 19 → 18.
Op2 on space:	459 → 459, 154 → 132.
Jaccard on space:	56 → 56, 130 → 129.
Tarantula on space:	56 → 56, 130 → 139.
Wong2 on space:	115 → 115, 368 → 296.
Wong1 on space:	725 → 725, 570 → 251.
Rogot1 on space:	78 → 78, 309 → 220.
Ochiai on space:	75 → 75, 122 → 93.
Cohen on space:	60 → 60, 153 → 123.
D* on space:	56 → 56, 130 → 109.

Tests on 1-fault programs

Op2 on print_tokens:	5 → 5.
Jaccard on print_tokens:	17 → 17.
Tarantula on print_tokens:	21 → 21.
Wong2 on print_tokens:	64 → 64.
Wong1 on print_tokens:	1 → 1.
Rogot1 on print_tokens:	55 → 55.
Ochiai on print_tokens:	9 → 9.
Cohen on print_tokens:	20 → 20.
D* on print_tokens:	17 → 17.
Op2 on print_token2:	3 → 3.
Jaccard on print_token2:	27 → 27.
Tarantula on print_token2:	28 → 28.
Wong2 on print_token2:	50 → 50.
Wong1 on print_token2:	1 → 1.
Rogot1 on print_token2:	40 → 40.
Ochiai on print_token2:	18 → 18.
Cohen on print_token2:	28 → 28.
D* on print_token2:	27 → 27.
Op2 on replace:	10 → 10.
Jaccard on replace:	23 → 23.
Tarantula on replace:	24 → 24.
Wong2 on replace:	77 → 77.
Wong1 on replace:	1 → 1.
Rogot1 on replace:	64 → 64.
Ochiai on replace:	15 → 15.
Cohen on replace:	24 → 24.
D* on replace:	23 → 23.
Op2 on schedule:	15 → 15.
Jaccard on schedule:	6 → 6.
Tarantula on schedule:	6 → 6.
Wong2 on schedule:	31 → 31.
Wong1 on schedule:	13 → 13.
Rogot1 on schedule:	35 → 35.
Ochiai on schedule:	6 → 6.
Cohen on schedule:	6 → 6.
D* on schedule:	6 → 6.
Op2 on schedule2:	49 → 49.
Jaccard on schedule2:	67 → 67.
Tarantula on schedule2:	67 → 67.
Wong2 on schedule2:	83 → 83.
Wong1 on schedule2:	1 → 1.
Rogot1 on schedule2:	83 → 83.
Ochiai on schedule2:	58 → 58.
Cohen on schedule2:	67 → 67.
D* on schedule2:	67 → 67.
Op2 on tcas:	11 → 11.
Jaccard on tcas:	13 → 13.
Tarantula on tcas:	13 → 13.
Wong2 on tcas:	24 → 24.
Wong1 on tcas:	3 → 3.
Rogot1 on tcas:	23 → 23.
Ochiai on tcas:	13 → 13.
Cohen on tcas:	13 → 13.
D* on tcas:	13 → 13.
Op2 on tot_info:	10 → 10.
Jaccard on tot_info:	26 → 26.
Tarantula on tot_info:	29 → 29.
Wong2 on tot_info:	69 → 69.
Wong1 on tot_info:	1 → 1.
Rogot1 on tot_info:	69 → 69.
Ochiai on tot_info:	20 → 20.
Cohen on tot_info:	28 → 28.
D* on tot_info:	26 → 26.
Op2 on flex:	814 → 814.
Jaccard on flex:	760 → 760.
Tarantula on flex:	760 → 760.
Wong2 on flex:	813 → 813.
Wong1 on flex:	760 → 760.
Rogot1 on flex:	742 → 742.
Ochiai on flex:	760 → 760.
Cohen on flex:	814 → 814.
D* on flex:	760 → 760.
Op2 on grep:	1020 → 1020.
Jaccard on grep:	1102 → 1102.
Tarantula on grep:	1195 → 1195.
Wong2 on grep:	1393 → 1393.
Wong1 on grep:	569 → 569.
Rogot1 on grep:	1375 → 1375.
Ochiai on grep:	1066 → 1066.
Cohen on grep:	1113 → 1113.
D* on grep:	1102 → 1102.
Op2 on gzip:	674 → 674.
Jaccard on gzip:	740 → 740.
Tarantula on gzip:	740 → 740.
Wong2 on gzip:	674 → 674.
Wong1 on gzip:	640 → 640.
Rogot1 on gzip:	685 → 685.
Ochiai on gzip:	740 → 740.
Cohen on gzip:	674 → 674.
D* on gzip:	697 → 697.
Op2 on sed:	555 → 555.
Jaccard on sed:	670 → 670.
Tarantula on sed:	623 → 623.
Wong2 on sed:	705 → 705.
Wong1 on sed:	626 → 626.
Rogot1 on sed:	654 → 654.
Ochiai on sed:	676 → 676.
Cohen on sed:	609 → 609.
D* on sed:	670 → 670.
Op2 on space:	266 → 266.
Jaccard on space:	300 → 300.
Tarantula on space:	274 → 274.
Wong2 on space:	860 → 860.
Wong1 on space:	316 → 316.
Rogot1 on space:	667 → 667.
Ochiai on space:	403 → 403.
Cohen on space:	287 → 287.
D* on space:	290 → 290.

(the end)

[8] B. Baudry, F. Fleurey, and Y. Le Traon. (2006). "Improving test suites for efficient fault localization". In *Proceedings of the 28th IEEE International Conference on Software Engineering (ICSE 2006)*, pages 82–91.

[9] H. Birgit and W. Franz. (2012). "Spectrum enhanced dynamic slicing for better fault localization". In *Frontiers in Artificial Intelligence and Applications (FAIA 2012)*, volume 242, pages 420–425.

[10] T.M. Chilimbi, B. Liblit, K. Mehra, A.V. Nori, and K. Vaswani. (2009). "Holmes: Effective statistical debugging via efficient path profiling". In *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE 2009)*, pages 34–44.

[11] H. Cleve and A. Zeller. (2005). "Locating causes of program failures". In *Proceedings of the 27th IEEE International Conference on Software Engineering (ICSE 2005)*, pages 342–352.

[12] N. Cliff. "Dominance statistics: Ordinal analyses to answer ordinal questions". *Psychological Bulletin (1993)*, 114(3), pages 494-509.

[13] J. Cohen. (1960). "A coefficient of agreement for nominal scales". *Educational and Psychological Measurement (1960)*, 20(1), pages 37–46.

[14] J. De Kleer, B. .C Williams. "Diagnosing multiple faults". *A*rtificial intelligence, 1987, 32(1), page 97–130.

[15] V. Debroy and W.E. Wong. (2009). "Insights on fault interference for programs with multiple bugs". In *Proceedings of the 20th International Symposium on Software Reliability Engineering (ISSRE 2009)*, pages 165–174.

[16] L.R. Dice. (1945). "Measures of the amount of ecologic association between species". *Ecology (1945)*, 26(3), pages 297-302.

[17] H. Do, S. Elbaum, and G. Rothermel. (2005). "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact". *Empirical Software Engineering (ESE 2005)*, 10(4), pages 405–435.

[18] J.M. Duarte, J.B. Santos, and L.C. Melo. (1999). "Comparison of similarity coefficients based on RAPD markers in the common bean". *Genetics and Molecular Biology (1999)*, 22(3), pages 427–432.

[19] B.S. Everitt. (1978). "Graphical techniques for multivariate data". *North-Holland (1978)*.

[20] J.L. Fleiss. (1965). "Estimating the accuracy of dichotomous judgments". *Psychometrika (1965)*, 30(4), pages 469–479.

[21] C. Fu and B.G. Ryder. (2007). "Exception-chain analysis: revealing exception handling architecture in Java server applications". In *Proceedings of the 29th IEEE International Conference on Software Engineering (ICSE 2007)*, pages 230–239.

[22] R. Greiner, B. A. Smith, R. W. Wilkerson. "A correction to the algorithm in Reiter's theory of diagnosis". Artificial Intelligence, 1989, 41(1), pages 79–88.

[23] C. Gong, Z. Zheng, W. Li, and P. Hao. (2012). "Effects of class imbalance in test suites: An Empirical Study of Spectrum-Based Fault Localization". In *Proceedings of the 2012 IEEE Conference on Computer Software and Application (COMPSAC 2012)*, pp.470-475.

[24] L. Gong, D. Lo, L. Jiang, and H. Zhang. (2012). "Diversity maximization speedup for fault localization". In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, pages 30–39.

[25] L.A. Goodman and W.H. Kruskal. "Measures of association for cross classification". *Journal of the American Statistical Association 1954*, 49(268), pages 732–764.

[26] D. Gopinath, R.N. Zaeem, and S. Khurshid. (2012). "Improving the effectiveness of spectra-based fault localization using specifications". In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, pages 40–49.

[27] N. Gupta, H. He, X. Zhang, and R. Gupta. (2005). "Locating faulty code using failure-inducing chops". In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 263–272.

[28] R. Gore and P.F Reynolds. (2012). "Reducing confounding bias in predicate-level statistical debugging metrics". In *Proceedings of the 34th IEEE International Conference on Software Engineering (ICSE 2012)*, pages 463–473.

[29] C. Gouveia, J. Campos, and R. Abreu. (2013). "Using html5 visualizations in software fault localization". In *Proceedings of 2013 First IEEE Working Conference on Software Visualization (VISSOFT 2013)*, pages 1–10.

[30] Y. Guo. (2017). "Localizing and fixing faults in SQL predicates". In *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST 2017)*, pages 555–556.

[31] R.W. Hamming. (1950). "Error detecting and error correcting codes". *Bell System Technical Journal (1950)*, 29(2), pages 147–160.

[32] Walter Hamscher, Luca Console, and Johan de Kleer (Eds.). 1992. Readings in model-based diagnosis. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[33] S. Hong, B. Lee, T. Kwak, Y. Jeon, B. Ko, Y. Kim, and M. Kim. (2015). "Mutation-based fault localization for real-world multilingual programs". In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, pages 464–475.

[34] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. (1998). "An empirical investigation of program spectra". *Acm Sigplan Notices (1998)*, 33(7), pages 147–160.

[35] W. Hogerle, F. Steimann, and M. Frenkel. (2014). "More debugging in parallel". In *Proceedings of International Symposium on Software Reliability Engineering (ISSRE 2014)*, pages 133–143.

[36] P. Jaccard. (1901). "Étude comparative de la distribution florale dans une portion des Alpes et des Jura". *Bulletin del la Socit Vaudoise des Sciences Naturelles (1901)*, 37, pages 547–579.

[37] D. Jeffrey, N. Gupta, and R. Gupta. (2008). "Fault localization using value replacement". In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 167–178.

[38] J.A. Jones and M.J. Harrold. (2005). "Empirical evaluation of the Tarantula automatic fault-localization technique". In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005)*, pages 273–282.

[39] J.A. Jones, M.J. Harrold, and J. Stasko. (2002). "Visualization of test information to assist fault localization". In *Proceedings of the 24th IEEE International Conference on Software Engineering (ICSE 2002)*, pages 467–477.

[40] J.A. Jones, J.F. Bowring, and M.J. Harrold. (2007). "Debugging in parallel". In *Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA 2007)*, pages 16–26.

[41] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo. (2017). "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system". In *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS 2017)*, pages 114–125.

[42] P.S. Kochhar, X. Xia, D. Lo, and S. Li. (2016). "Practitioners' expectations on automated fault localization". In *Proceedings of the 2016 International Symposium on Software Testing and Analysis (ISSTA 2016)*, pages 165–176.

[43] E.F. Krause. (1973). "Taxicab geometry". *Mathematics Teacher (1973)*, 66(8), pages 695–706.

[44] G. Laghari, A. Murgia, and S. Demeyer. (2016). "Fine-tuning spectrum based fault localisation with frequent method item sets". In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, pages 274—285.

[45] H.J. Lee, L. Naish, and K. Ramamohanarao. (2009). "Study of the relationship of bug consistency with respect to performance of spectra metrics". In *Proceedings of the 2nd IEEE International Conference on Computer Science and Information Technology (ICCSIT 2009)*, pages 501–508.

[46] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, and M.I. Jordan. (2005). "Scalable statistical bug isolation". *Acm Sigplan Notices (2005)*, 40(6), pages 15–26.

[47] H. Li, Y. liu, Z. Zhang, and J. Liu. (2014). "Program structure aware fault localization". In *Proceedings of International Workshop on Innovative Software Development Methodologies and Practices (INNOSWDEV 2014), in conjunction with the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 40–48.

[48] X. Li, M. d'Amorim, and A. Orso. (2016). "Iterative user-driven fault localization". In *Proceedings of Haifa Verification Conference*, pages 82-–98.

[49] C. Liu, L. Fei, X. Yan, J. Han, and S.P. Midkiff. (2006). "Statistical debugging: A hypothesis testing-based approach". *IEEE Transactions on Software Engineering (TSE 2006)*, 32(10), pages 831–848.

[50] F. Lourenco, V. Lobo, and F. Bacão. (2004). "Binary-based similarity measures for categorical data and their application in self-organizing maps".

[51] A.D.S. Meyer, A.A.F. Garcia, A.P.D. Souza, and C.L.D Souza Jr. (2004). "Comparison of similarity coefficients used for cluster analysis with dominant markers in maize (Zea mays L)". *Genetics and Molecular Biology (2004)*, 27(1), pages 83–91.

[52] A.T. Nguyen, T.T. Nguyen, T.N. Nguyen, D. Lo, and C. Sun. (2012). "Duplicate bug report detection with a combination of information retrieval and topic modeling". In *Proceedings of the 27th IEEE/ACM*

International Conference on Automated Software Engineering (ASE 2012), pages 70–79.

[53] H. D. T. Nguyen, Q. Qi, A. Roychoudhury, and S, Chandra. (2013). "DSemFix: Program Repair via Semantic Analysis". In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, pages 772–781.

[54] B. Liu, S. Nejati, and L.C. Briand. (2017). "Improving fault localization for Simulink models using search-based testing and prediction models". In *Proceedings of the 34th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2017)*, pages 359–370.

[55] Y. Liu, Z. Li, L. Wang, Z. Hu, and R. Zhao. (2017). "Statement-oriented mutant reduction strategy for mutation based fault localization". In *Proceedings of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS 2017)*, pages 126–137.

[56] X. Liu, Y. Liu, Z. Li, and R. Zhao. (2017). "Fault classification oriented spectrum-based fault localization". In *Proceedings of the 41st IEEE Annual Computer Software and Applications Conference (COMPSAC 2017)*, pages 256–261.

[57] H.L. Lu, R. Gao, S.K. Huang, and W.E. Wong. (2016). "Spectrum-base fault localization by exploiting the failure path". In *Proceedings of the 2016 International Computer Symposium (ICS 2016)*, pages 252–257.

[58] W. Masri and R.A. Assi. (2014). "Prevalence of coincidental correctness and mitigation of its impact on fault localization". *ACM Transactions on Software Engineering and Methodology (TOSEM 2014)*, 23(1), 8.

[59] S. Mechtaev, J. Yi, A. Roychoudhury. (2015). "DirectFix: Looking for Simple Program Repairs". In *Proceedings of IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE 2015)*, pages 504–508.

[60] L. Naish, H.J. Lee, and K. Ramamohanarao. (2011). "A model for spectra-based software diagnosis". *ACM Transactions on Software Engineering Methodology (TOSEM 2011)*, 20(3), 11.

[61] L. Naish and K. Ramamohanarao. (2015). "Multiple bug spectral fault localization using genetic programming". In *Proceedings of the 24th Australasian Software Engineering Conference (ASWEC2015)*, pages 11–17.

[62] N. Neelofar, L. Naish, and K. Ramamohanarao. (2017). "Spectral-based fault localization using hyperbolic function". *Software Practice & Experience (SPE 2017)*, 2017(3).

[63] A. Ochiai. (1957). "Zoogeographic studies on the solenoid fishes found in Japan and its neighbouring regions". *Bulletin of the Japanese Society for Fish Science (1957)*, 22(9), pages 526–530.

[64] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M.D. Ernst, D. Pang, and B. Keller. (2017). "Evaluating and improving fault localization". In *Proceedings of the 39th IEEE International Conference on Software Engineering (ICSE 2017)*, pages 609-620.

[65] C. Parnin and A. Orso. (2011). "Are automated debugging techniques actually helping programmers?". In *Proceedings of the 2011 international symposium on software testing and analysis (ISSTA 2011)*, pages 199–209.

[66] F. Pastore and L. Mariani. (2013). "Ava: Supporting debugging with failure interpretations". In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation (ICST 2013)*, pages 416—421.

[67] I. Pill and T. Quaritsch, "RC-Tree: A variant avoiding all the redundancy in Reiter's minimal hitting set algorithm," In *Proceedings of 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Gaithersburg, MD, 2015, pages 78–84.

[68] I. Pill and F. Wotawa. "Exploiting Observations from Combinatorial Testing for Diagnostic Reasoning". In *Proceedings of 30th International Workshop on Principles of Diagnosis*, DX2019.

[69] A. Perez, R. Abreu, and A. van Deursen. (2017). "A test-suite diagnosability metric for spectrum-based fault localization approaches". In *Proceedings of the 39th IEEE International Conference on Software Engineering (ICSE 2017)*, pages 654–664.

[70] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. (2012). "Darwin: An approach to debugging evolving programs". *ACM Transactions on Software Engineering and Methodology (TOSEM 2012)*, 21(3), 19.

[71] Y. Qi, X. Mao, Y. Lei, and C. Wang. (2013). "Using automated program repair for evaluating the effectiveness of fault localization techniques". In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2012)*, pages 191—201.

[72] R. Reiter. "A theory of diagnosis from first principles". Artificial intelligence, 1987, 32(1), page 57–95.

[73] M. Renieres, S. P. Reiss. "Fault localization with nearest neighbor queries". In *Proceedings of18th IEEE International Conference on Automated Software Engineering (ICSE 2003)*, pages 30–39.

[74] T. Reps, T. Ball, M. Das, and J. Larus. (1997). "The use of program profiling for software maintenance with applications to the year 2000 problem". *ACM Software Engineering Notes*, 1301(6), pages 432–449.

[75] D.J. Richardson and M.C. Thompson. (1993). "An analysis of test data selection criteria using the relay model of fault detection". *IEEE Transactions on Software Engineering (TSE 1993)*, 19(6), pages 533–553.

[76] D.J. Rogers and T.T. Tanimoto. (1960). "A computer program for classifying plants". *Science (1960)*, 132(3434), pages 1115–1118.

[77] E. Rogot and I.D. Goldberg. (1966). "A proposed index for measuring agreement in test-retest studies". *Journal of Chronic Diseases (1966)*, 19(9), pages 991–1006.

[78] P.F. Russel and T.R. Rao. (1940). "On habitat and association of species of Anopheline larvae in south-eastern Madras". *Journal of Malarial Institute of India (1940)*, 3(1), pages 153–178.

[79] R.K. Saha, M. Lease, S. Khurshid, and D.E. Perry. (2013). "Improving bug localization using structured information retrieval". In *Proceedings of the 28th International Conference on Automated Software Engineering (ASE 2013)*, pages 345–355.

[80] R. Santelices, J.A. Jones, Y. Yu, and M.J. Harrold. (2009). "Lightweight fault-localization using multiple coverage types". In *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE 2009)*, pages 56–66.

[81] F. Servant and J. A. Jones. (2011). "History slicing". In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 452–455.

[82] F. Servant and J. A. Jones. (2012). "History slicing: assisting code-evolution tasks". In *Proceedings ofthe ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012)*, pages 43–53.

[83] W.A. Scott. (1955). "Reliability of content analysis: the case of nominal scale coding". *Public Opinion Quarterly (1955)*, 19(3), pages 321–325.

[84] J. Sohn and S. Yoo. (2017). "FLUCCS: Using code and change metrics to improve fault localization". In *Proceedings of the 26th International Symposium on Software Testing and Analysis (ISSTA 2017)*, pages 273–283.

[85] s. Song. (2014). "Estimating the effectiveness of spectrum-based fault localization". In *Proceedings of the 22nd ACM Sigsoft International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 814-816.

[86] F. Steimann, M. Frenkel, and R. Abreu. (2013). "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators". In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*, pages 314–324.

[87] N. DiGiuseppe and James A. Jones. (2011). "Fault interaction and its repercussions". In *Proceedings of the 27th IEEE international conference on software maintenance (ICSM 2011)*, pp. 3-12. IEEE, 2011.

[88] C.M. Tang, W.K. Chan, and Y.T. Yu. (2014). "Extending the theoretical fault localization effectiveness hierachy with empirical results at different code abstraction levels". In *Proceedings of the 38th Annual IEEE International Computer Software and Applications Conference (COMPSAC 2014)*, pages 161-170.

[89] C.M. Tang, W.K. Chan,Y.T. Yu, and Z. Zhang. (2017). "Accuracy graphs of spectrum-based fault localization formulae". *IEEE Transactions on Reliability (TRel 2017)*, 66(2), pages 621–637.

[90] C.M. Tang, W.K. Chan, and Y.T. Yu. (2017). "Theoretical, weak and strong accuracy graphs of spectrum-based fault localization formulae". In *Proceedings of the 41st Annual Computer Software and Applications Conference (COMPSAC 2017)*, pages 78–83.

[91] C.M. Tang, J. Keung, Y.T. Yu, and W.K. Chan. (2016). "DFL: Dual-service fault localization". In *Proceedings of the 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS 2016)*, pages 412-422.

[92] I. Vessey. (1986). "Expertise in debugging computer programs: An analysis of the content of verbal protocols". *IEEE Transactions on Systems, Man, and Cybernetics*, 16(5), pages 621–637.

[93] K. Wagstaff, C. Cardie, S. Rogers, and Stefan Schrödl. (2001). "Constrained K-means Clustering with Background Knowledge".

In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001)*, pages 577–584.

[94] Q. Wang, C. Parnin, and A. Orso. (2015). "Evaluating the usefulness of ir-based fault localization techniques". In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*, pages 1—11.

[95] X. Wang, S.C. Cheung, W.K. Chan, and Z. Zhang. (2009). "Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization". In *Proceedings of the 31st IEEE International Conference on Software Engineering (ICSE 2009)*, pages 45–55.

[96] X. Wang and Y. Liu. (2016). "Fault localization using disparities of dynamic invariants". *Journal of Systems and Software (JSS 2016)*, vol. 122, pages 144—154,.

[97] Y. Wang and Z. Huang. (2016). "Weighted control flow subgraph to support debugging activities". In *Proceedings of 2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C 2016)*, pages 131—134.

[98] Jin, Wei and Orso, Alessandro. (2013). "F3: Fault localization for field failures". In *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*, pages 213–223.

[99] W.E. Wong, V. Debroy, R. Gao, and Y. Li. (2014). "The DStar method for effective software fault localization". *IEEE Transactions on Reliability (TRel 2014)*, 63(1), pages 290–308.

[100] W.E. Wong, Y. Qi, L. Zhao, and K.Y. Chai. (2007). "Effective fault localization using code coverage". In *Proceedings of the 31st Annual International Conference on Computer Software and Applications (COMPSAC 2007)*, pages 449–456.

[101] W.E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. (2016). "A survey on software fault localization". IEEE Transactions on Software Engineering (TSE 2016), 42(8), pages 707–740.

[102] X. Xie, F. C. Kuo, T. Y. Chen, S. Yoo, and M. Harman. (2013) "Provably optimal and human-competitive results in SBSE for spectrum based fault localisation." In *International Symposium on Search Based Software Engineering (SBSE 2013)*, pages 224–238.

[103] X. Xie, T. Y. Chen, F. Kuo, and B. Xu. (2013). "A theoretical analysis of the risk evaluation formulae for spectrum-based fault localization". *ACM Transactions on Software of Engineering and Methodology (TOSEM 2013)*, 22(4), 31.

[104] J. Xu, Z. Zhang, W.K. Chan, T.H. Tse, and S. Li. (2013). "A general noise-reduction framework for fault localization of java programs". *Information and Software Technology*, 55(5), pages 880 – 896.

[105] J. Xuan and M. Martin. (2014). "Test case purification for improving fault localization". In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 52–63.

[106] M. D. Yang, Y. Qi, X. Mao X. (2017). "An Empirical Study on the Usage of Fault Localization in Automated Program Repair". In *Proceedings of IEEE International Conference on Software Maintenance & Evolution (ICSME 2017)*.

[107] S. Yoo, M. Harman, and D. Clark. (2013). "Fault localization prioritization: Comparing information-theoretic and coverage-based approaches". *ACM Transactions on Software Engineering and Methodology (TOSEM 2013)*, 22(3), 9.

[108] S. Yoo, X. Xie, F.C. Kuo, T.Y. Chen and M. Harman. (2017). "Human competitiveness of genetic programming in spectrum-based fault localisation: theoretical and empirical analysis". *ACM Transactions on Software Engineering and Methodology (TOSEM 2017)*, 26(1), 4.

[109] K. Yu, M. Lin, Q. Gao, H. Zhang, and X. Zhang. (2011). "Locating faults using multiple spectra-specific modes". In *Proceedings of the 2011 ACM Symposium on Applied Computing (2011)*, pages 1404–1410.

[110] D. Yuan, H. Mai, W. Xiong, L. Tan, L. Zhou, and S. Pasupathy. (2010). "SherLog: error diagnosis by connecting clues from runtime logs". *ACM SIGARCH computer architecture news (2010)*, 38(1), pages 143–154.

[111] A. Zeller. (1997). "Yesterday, my program worked. Today, it does not. Why?.". In *Proceedings of the 7th European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 1997)*, pages 253-267.

[112] Z. Zhang, B. Jiang, W.K. Chan, and T.H. Tse. (2008). "Debugging through evaluation sequences: A controlled experimental study". In *Proceedings of the 32nd Annual International Computer Software and Applications Conference (COMPSAC 2008)*, pages 128–135.

[113] Z. Zhang, W.K. Chan, T.H. Tse, B. Jiang, and X. Wang. (2009). "Capturing propagation of infected program states". In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE 2009)*, pages 43–52.

[114] Z. Zhang, W.K. Chan, T.H. Tse, Y.T. Yu, and P. Hu. (2011). "Non-parametric statistical fault localization". *Journal of Systems and Software (JSS 2011)*, 84(6), pages 885–905.

[115] Z. Zhang and J. Zhang. (2011). "Characterizing failure-causing parameter interactions by adaptive testing". In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA 2011)*, pages 331–341.

[116] Z. Zhang, W.K. Chan, and T.H. Tse. (2012). "Fault localization based only on failed runs". *IEEE Computer (COMPUTER 2012)*, 45(6), pages 64–71.

[117] L. Zhang, L. Yan, Z. Zhang, J. Zhang, W.K. Chan, and Z. Zhang. (2017). "A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization". *Journal of System and Software (JSS 2017)*, 129, pages 174–187.

[118] X.Y. Zhang, Z. Zheng, and K.Y. Cai. (2017). "Exploring the usefulness of unlabelled test cases in software fault localization". *Journal of Systems and Software (JSS 2017)*, 136, pages 1–13.

[119] M. Zhang, X. Li, L. Zhang, and S. Khurshid. (2017). "Boosting spectrum-based fault localization using PageRank". In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2017)*, pages 261-272.

[120] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso. (2014). "Mimic: Locating and understanding bugs by analyzing mimicked executions". In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE 2014)*, pages 815–826.

PLACE PHOTO HERE

Long Zhang got his PhD degree at the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. He obtained his bachelor degree in Computer Science and Technology from Hefei University of Technology. His research interests include software testing and AI.

PLACE PHOTO HERE

Zijie Li is a master student at the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. He obtained his bachelor degree from Tsinghua University. His research interests include program debugging and program repair.

PLACE PHOTO HERE

Yang Feng is an assistant researcher at the Department of Computer Science and Technology, Nanjing University, Nanjing, China. He got his PhD degree in Software Engineering at University of California, Irvine, advised by Prof. James. A. Jones. He obtained his bachelor degree and master degree in software engineering at Nanjing University, China. His research focuses on the areas of the program comprehension, testing, debugging, program analysis, and crowdsourced software engineering.

PLACE
PHOTO
HERE

Zhenyu Zhang is an associate professor at the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. He obtained his PhD degree from The University of Hong Kong, and master and bachelor degree from Tsinghua University. His research interests are testing and debugging for software and systems. He has published research results in venues such as Computer, TSE, TSC, TRel, ICSE, FSE, ASE, and WWW.

PLACE
PHOTO
HERE

W.K. Chan is an associate professor at City University of Hong Kong. His current main research interest is program analysis and testing for concurrent software and systems. He is the Special Issues Editor of Journal of Systems and Software. He has published more than 100 papers in venues such as TOSEM, TSE, TPDS, TSC, TRel, CACM, Computer, ICSE, FSE, ISSTA, ASE, WWW, ICWS, and ICDCS.

PLACE
PHOTO
HERE

Jian Zhang is a research professor at the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, and also a professor at the University of Chinese Academy of Sciences. His current research interests include program analysis, combinatorial testing, automated reasoning and constraint solving. He is an editorial board member of Chinese Journal of Computers, Scientia Sinica Informationis, IEEE Transactions on Reliability, Journal of Computer Science and Technology, Frontiers of Computer Science, and Journal of Frontiers of Computer Science and Technology.

PLACE
PHOTO
HERE

Yuming Zhou received the Ph.D. degree in computer science from Southeast University in 2003. From January 2003 to December 2004, he was a researcher at Tsinghua University. From February 2005 to February 2008, he was a researcher at Hong Kong Polytechnic University. He is currently a professor in the Department of Computer Science and Technology at Nanjing University. His main research interests are empirical software engineering.