International Journal of Software Engineering
and Knowledge Engineering
Vol. 21, No. 6 (2011) 803–827

DOI: 10.1142/S0218194011005505

TIES WITHIN FAULT LOCALIZATION RANKINGS: EXPOSING AND ADDRESSING THE PROBLEM

XIAOFENG XU[*,†,§], VIDROHA DEBROY[†,¶], W. ERIC WONG[†,‖]
and DONGHUI GUO[*,‡,**]

[*]*Department of Physics, Xiamen University, China*

[†]*Department of Computer Science*
University of Texas at Dallas, USA

[‡]*School of Information Science and Technology*
Xiamen University, China
[§]*xxfeng@xmu.edu.cn*
[¶]*vxd024000@utdallas.edu*
[‖]*ewong@utdallas.edu*
[**]*dhguo@xmu.edu.cn*

Software fault localization techniques typically rank program components, such as statements or predicates, in descending order of their suspiciousness (likelihood of being faulty). During debugging, programmers may examine these components, starting from the top of the ranking, in order to locate faults. However, the assigned suspiciousness to each component may not always be unique, and thus some of them may be tied for the same position in the ranking. In such a scenario, the total number of components that a programmer needs to examine in order to find the faults may vary considerably. The greater the variability, the harder it is for a programmer to decide which component to examine first, and the harder it is to accurately compute the expected effectiveness of a fault localization technique. In this paper, we first conduct a case study, based on three fault localization techniques across four sets of programs, which reveals that the phenomenon of assigning the same suspiciousness to multiple components is not limited to any technique or program in particular. Thus, to reduce variability and alleviate this problem, four tie-breaking strategies are discussed and evaluated empirically in our second case study. Results indicate that the strategies can not only reduce the number of ties in the rankings, but also maintain the effectiveness of the fault localization techniques. We also propose a new metric for evaluating fault localization techniques called CScore, which takes the notion of ties into account. Finally, an additional slicing-based approach to breaking ties is discussed briefly, which aims to provide further insights into tie-breaking and stimulate further research in the area.

Keywords: Program debugging; fault localization; critical ties; suspiciousness; tie-breaking.

[*]Xiaofeng Xu is a PhD student at Xiamen University under the supervision of Professor Donghui Guo and a visiting student at the University of Texas at Dallas from August 2009 to August 2011 under Professor W. Eric Wong. This research is sponsored by the China Scholarship Council (CSC), the National Natural Science Foundation of China (Grant No. 60940029), the PhD program Foundation of Ministry of Education of China, and the University of Texas at Dallas.

1. Introduction

Among all the program debugging activities, software fault localization (hereafter referred to simply as fault localization) has been recognized to be one of the most expensive [14]. This realization has catalyzed the proposal and development of many fault localization techniques over recent years [1, 2, 7, 9, 11, 12, 17–21, 24]. Several such techniques use execution traces collected at runtime during program testing, as well as test execution results (whether an execution succeeds or fails) to compute the suspiciousness of each statement.[a] This 'suspiciousness' represents the likelihood of the corresponding statement being faulty. Once the statements have been ranked in descending order of their suspiciousness, from most suspicious to least, they are examined one-by-one starting from the top of the ranking until a faulty statement is identified. A good fault localization technique should place a faulty statement towards the top of its ranking, if not at the very top. Ideally, a faulty statement should have a much higher suspiciousness than a non-faulty statement.

However, the suspiciousness of a statement assigned by a fault localization technique is not always unique when compared with the suspiciousness of other statements. Multiple statements may be assigned the same suspiciousness and therefore are tied for the same position in the ranking. The existence of such tied statements implies that the fault localization technique in question cannot distinguish these statements from one another in terms of their likelihood of being faulty. Programmers are offered no guidance on what to examine first, given a set of statements that are tied, and are thus forced to rely on their own judgment or intuition.

For the purpose of discussion, let us consider the case where in a ranking there is a group of statements with the same suspiciousness and only one of them is faulty. In the very best case, the programmer shall examine the faulty statement first and there will be no need to examine other statements in the group, as we would stop examining after finding the fault. But in the worst case, we would examine all non-faulty statements in the group before we could reach the faulty statement. Thus, in terms of the number of statements that need to be examined to find the fault, we have two different levels of effectiveness — the *best* and the *worst*. The actual effectiveness may be anywhere in between the best and the worst. The larger the gap between these two levels of effectiveness (the more statements that are tied with the faulty statement), the more difficult it is to accurately estimate what the actual effectiveness would be. Thus, ties are undesirable in a ranking of program statements.

An important question that then arises is if ties between statements are a common occurrence, and if they are prevalent regardless of which fault localization techniques

[a] Fault localization techniques can be used to locate different types of faulty program components. For the purposes of this paper we consider program components to be statements with the understanding that they could have just as easily been other components such as functions, blocks, predicates, etc.

or subject programs are used. We take this opportunity to point out that for the purposes of this paper, ties between groups of statements where none of them is faulty are considered immaterial. Further discussion on this is presented in Sec. 2.1. To continue, if a faulty statement is not expected to be tied with other non-faulty statements, then why would we need to worry about ties? To address such concerns, experiments were performed using three fault localization techniques (Tarantula [7], Ochiai [1], and Heuristic III [17]) across four sets of programs (the Siemens Suite, and the grep, gzip and make programs). Results indicate that ties involving faulty statements are quite frequent, and they occur regardless of the choice of fault localization technique or subject program. Then the immediate question that arises is how these ties might be broken such that the problem mentioned above is alleviated, and yet the effectiveness of a fault localization technique is not affected adversely. To this effect, four tie-breaking strategies are discussed and evaluated in our paper.

The contributions of this paper can be summarized as:

(1) Using three fault localization techniques (Tarantula [7], Ochiai [1], and Heuristic III [17]) on different programs (the Siemens suite, grep, gzip and make), the existence and scale of the problems involving ties is revealed. Furthermore, we show that this concern is not limited to any technique or subject program in particular.
(2) Four strategies are proposed to break ties that can be applied on top of pre-existing fault localization techniques, and do not require them to be modified in any way.
(3) Empirical data shows that some of the strategies can break ties among statements to a considerable extent, yet do so without having an adverse impact on fault localization effectiveness.

The remainder of this paper is organized as follows: Sec. 2 discusses in detail the problem that is addressed and describes the fault localization techniques and subject programs evaluated. Empirical evidence is presented to illustrate the scale of the problem. Section 3 explains four tie-breaking strategies designed to alleviate the problem, which are subsequently evaluated in Sec. 4. Section 5 provides a discussion on some of the relevant issues and Sec. 6 reports potential threats to validity. Related work is reviewed in Sec. 7, and we give our conclusions in Sec. 8.

2. Illustrating the Problem

In this section we further discuss ties among statements in the rankings that are produced by fault localization techniques, and investigate via case studies, the degree to which this may occur. For each of the fault localization techniques that are used, a succinct overview is also provided.

2.1. *Ties and critical ties*

There is a subtle distinction between the types of ties that may occur among statements in a ranking. While it is possible that a faulty statement may be tied with non-faulty statements, it is also quite likely that non-faulty statements are tied to one another for the same position in a ranking. The former scenario is of greater concern to us than the latter. To better understand why, let us again consider a hypothetical ranking where only one of the statements is faulty. Given that there may be a set of statements S none of which is faulty, yet they are all assigned the same suspiciousness, we can construct two scenarios: First, it is possible the statements in S have a higher suspiciousness than the actual faulty statement. In this case, every statement in S will have to be examined before the faulty statement regardless of the internal order in which the statements of S are examined. Thus, breaking ties for statements in S is of no immediate gain with respect to the total fault localization effectiveness in terms of the number of statements that must be examined to find the fault.

Now consider the case where the statements in S are assigned a suspiciousness that is less than that of the faulty statement. In this scenario, the faulty statement will be examined before the statements in S, and once we have located the faulty statement, we have no reason to continue examining the ranking. Therefore, again the internal order of examination of the statements in S is irrelevant with regards to the fault localization effectiveness. The effectiveness in fact only undergoes change when ties are broken with respect to a group of statements, all of which have the same assigned suspiciousness, that contains the faulty statement. Thus, even though the number of ties for groups that do not contain the faulty statement may have been altered, when evaluating the tie-breaking strategies, which will be presented in Sec. 3, we disregard the change in ties for such groups and focus only on tie-breaks among groups that contain faulty statements. To facilitate subsequent discussion, we present two definitions:

Definition 1. Ties. A Tie T is defined as a set of statements, each of which has been assigned the same suspiciousness, and therefore share the same position in a ranking, with respect to the fault localization technique used.

Definition 2. Critical Ties and critically tied statements. A Critical Tie CT is a tie which contains a faulty statement. The statements in a critical tie are called critically tied statements.

2.2. *Fault localization techniques*

We utilize three fault localization techniques for our experiments — Tarantula [7], Ochiai [1], and Heuristic III [17]. Each one of these techniques was chosen with a specific reason in mind: Tarantula because it is simple and popular, and Ochiai and Heuristic III because they have reported better fault localization effectiveness than Tarantula, yet are quite different from one another in terms of their design.

2.2.1. *Tarantula*

Tarantula [7] is a fault localization technique that follows the intuition that statements which are executed primarily by more failed test cases are highly likely to be faulty. Additionally, statements that are executed primarily by more successful test cases are less likely to be faulty. Tarantula thus assigns a suspiciousness value to each statement based on the following formula:

$$susp(s) = \frac{\frac{failed(s)}{totalfailed}}{\frac{failed(s)}{totalfailed} + \frac{successful(s)}{totalsuccessful}} \tag{1}$$

where $susp(s)$ is the suspiciousness of statement s; $successful(s)$ and $failed(s)$ are the number of successful and failed test cases[b] that execute s respectively; and *totalsuccessful* and *totalfailed* are the total number of successful and failed test cases, respectively.

2.2.2. *Ochiai*

Ochiai [1] is a similarity coefficient which can be used for fault localization. Ochiai assigns a suspiciousness value to each statement based on the following formula:

$$susp(s) = \frac{failed(s)}{\sqrt{totalfailed \times (failed(s) + successful(s))}} \tag{2}$$

where *susp(s)*, *successful(s)*, *failed(s)*, and *totalfailed* are the same as in Eq. (1).

2.2.3. *Heuristic III*

Heuristic III [17] is a fault localization technique that posits that all the failed, and successful, test cases should not have the same contribution in computing the suspiciousness of a statement. It thus divides failed test cases that execute the same statement into several groups such that each group contains a certain number of failed tests. The same also applies to the successful test cases. Different contribution weights are assigned to each failed or successful group. Heuristic III assigns a suspiciousness value to each statement based on the following formula:

$$susp(s) = \sum_{i=1}^{S_F} w_{F_i} \times n_{F_i} - \sum_{i=1}^{S_S} w_{S_i} \times n_{S_i} \tag{3}$$

where S_F and S_S represent the number of failed and successful groups, n_{Fi} and n_{Si} are the number of failed and successful test cases in the ith group, and w_{Fi} and w_{Si}

[b]For our purposes a successful test case is one where the observed output/behavior of an execution is the same as the expected output/behavior, and by the same token, a failed test case is one where the observed output/behavior of an execution deviates from the expected output/behavior. Note also the terms "*passed*" and "*totalpassed*" used in [7] are the same as "*successful*" and "*totalsuccessful*" in this paper.

are the contribution weights of the ith failed and successful groups, respectively. As per the experiments in [17], we use $S_F = S_S = 3$, $n_{F_1} = n_{S_1} = 2$, and $n_{F_2} = n_{S_2} = 4$, implying the third failed group contains all the remaining failed tests and the third successful group contains all the remaining successful tests. Also we set $w_{F_1} = 1$, $w_{F_2} = 0.1$, $w_{F_3} = 0.01$, $w_{S_1} = 1$, $w_{S_2} = 0.1$ and $w_{S_3} = \alpha \times \chi_{F/S}$, where $\alpha = 0.001$ and $\chi_{F/S}$ is the ratio of total number of failed and total number of successful test cases. Readers interested in further details of this technique are referred to [17].

2.3. *Subject programs and data collection*

Four sets of programs (both small and large size) are used in our experiments — the programs of the Siemens suite, and the grep, gzip, and make programs.

Siemens Suite Programs: The Siemens suite has been used extensively in fault localization related studies [3, 6, 7, 9, 10, 17, 18, 22, 26]. The suite contains seven correct versions (one for each program), 132 faulty versions and the corresponding test cases. They are downloaded from [13]. In our study, three faulty versions are excluded: version 9 of 'schedule2' because it does not result in any test case failure, and versions 4 and 6 of 'print _tokens' because these faults are located in the header files and not in the .c files. Thus, we have 129 faulty versions in total.

Gzip: Version 1.1.2 of the Gzip program, with 16 seeded bugs and 217 test cases, is downloaded from [4]. Nine faults are excluded from our study since no failure is observed for these versions, and six test cases are discarded because they cannot be executed in our environment. Following a fault injection approach similar to that which is proposed in [9], an additional 21 bugs are injected by us. This means that a total of 28 faulty versions are used as per the gzip program.

Grep: Version 2.2 of the Grep program is downloaded from [4] along with 18 faulty versions and a suite of 470 test cases. Fourteen faults are excluded from our study as no failure is observed. Two additional bugs injected by the authors of [9] are also used. Following the same approach mentioned above, an additional 13 bugs are injected by us bringing the total number of faulty versions to 19.

Make: Version 3.76.1 of make is downloaded from [4] together with 793 test cases and 19 fault versions. Of these, 15 faults which cannot be detected by the downloaded test cases are excluded in our study, and in a manner similar to the other programs, we injected an additional 27 bugs for a total of 31 usable faulty versions.

Table 1 summarizes each program in terms of the number of faulty versions used, test cases, and size (executable statements[c]). Note that the programs in the first seven rows (i.e., print_tokens to tot_info) all belong to the Siemens suite. Also while each faulty version contains only one fault, the strategies proposed in this

[c] Multi-line statements are combined into one source line so that they are counted as one during an execution, and code such as blank lines, comments, function and variable declarations are omitted from consideration as per the discussion presented in [7]. What we are then left with are executable statements.

Table 1. Summary of subject programs.

Program	Num. faulty versions used	Num. test cases	Num. executable statements
print_tokens	5	4130	175
print_tokens2	10	4115	178
schedule	9	2650	121
schedule2	9	2710	112
replace	32	5542	216
tcas	41	1608	55
tot_info	23	1052	113
gzip	28	211	1670
grep	19	470	3306
make	31	793	5318

paper can easily be applied to handle programs with multiple faults as well, which is further discussed in Sec. 6.

As far as the data collection step is concerned, the programs in the Siemens suite were executed on a PC with a SunOS 5.10 (Solaris 10) operating system and version 3.4.3 of the gcc compiler. Grep, gzip, and make were executed under the same operating system except with gcc 3.4.4 as the compiler. Each faulty version was executed against all its corresponding available test cases and all the execution traces were collected using a revised version of χSuds [23]. As mentioned in footnote b, for each test case execution, if the outputs of the faulty version were the same as on the correct version, the execution was determined to be 'successful', and if not, then the execution was declared as 'failed'.

2.4. *The scale of the tie-related problems*

Prior to solving a problem, one must first decide whether the problem itself is worth fixing. Recall our discussion from Sec. 1, where we asked ourselves if ties in the rankings (critical ties for our purposes) were even worth breaking. To answer this, we need to determine the scale of the tie-related problems. The term 'scale' is used in a general sense and encompasses multiple factors such as the frequency of occurrence of critical ties, and the size of a critical tie. Also investigated is the question of whether the presence or absence of critical ties has something to do with the choice of fault localization technique or the subject programs under study. We address these issues through empirical data collected using the fault localization techniques in Sec. 2.2 across all the faulty versions of the subject programs in Sec. 2.3. Figures 1–4 (corresponding to the Siemens, gzip, grep and make programs, respectively) present this data as box plots. Each of the box plots represents the chosen fault localization techniques on the x-axis and provides the ratio of — the size of the critical tie[d] to the total number of executable statements in the program — on the

[d]We can only have one critical tie per faulty version as each of the faulty versions studied has a single fault in it, and for a fault that spans multiple lines, the examination of a ranking stops as soon as the first statement corresponding to the fault is located.

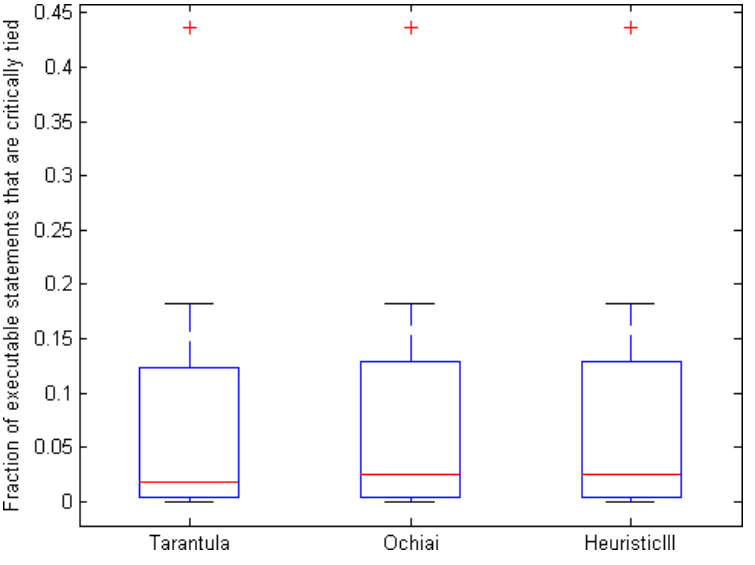

Fig. 1. Fraction of program that is critically tied for the Siemens suite.

y-axis with respect to each of the techniques. They present the fraction of the program that is critically tied.

We are immediately able to draw several conclusions based on these box plots. First, we observe that all of the studied fault localization techniques produce critical

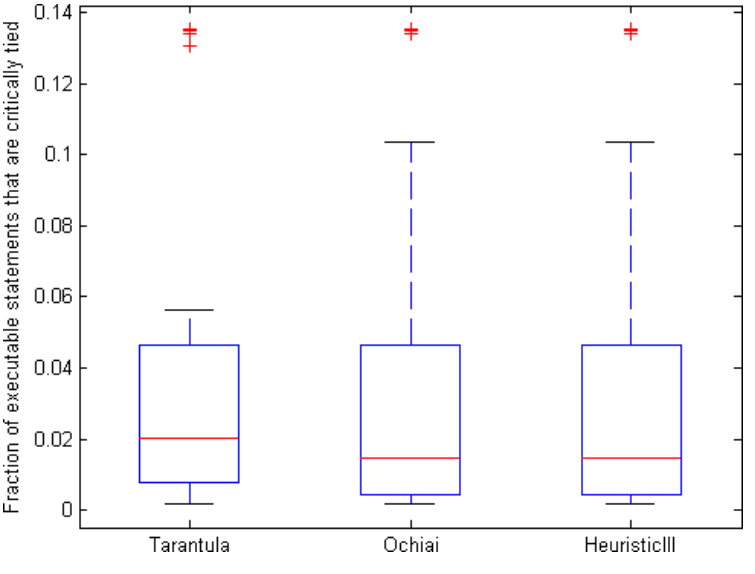

Fig. 2. Fraction of program that is critically tied for gzip.

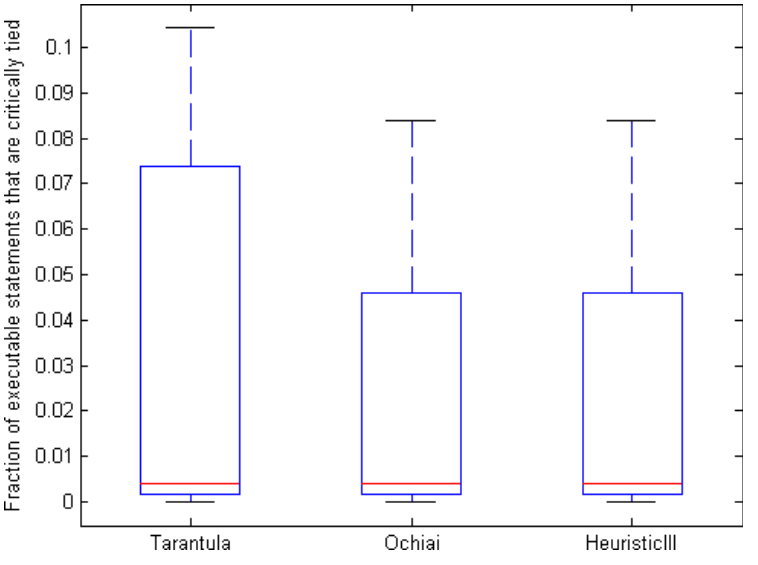

Fig. 3. Fraction of program that is critically tied for grep.

ties, and furthermore this is true regardless of which subject program is under consideration. With regard to the scale or significance of the critical ties, we observe that on the Siemens suite programs (Fig. 1), the maximum recorded fraction is almost as high as 0.2 or 20%, which means that for at least one of the faulty versions,

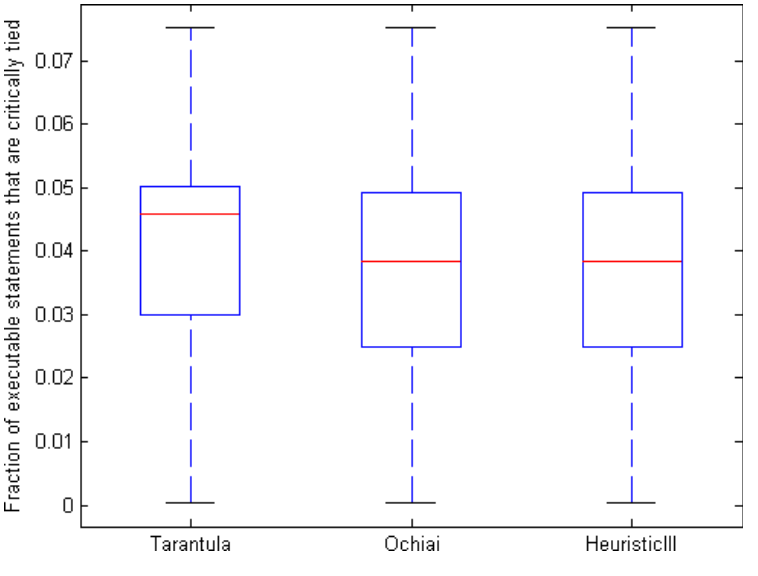

Fig. 4. Fraction of program that is critically tied for make.

almost 20% of the code (executable statements) is critically tied. This is the case for all three of the fault localization techniques studied. Furthermore, if we consider the maximum outliers, then we observe that almost 45% of the code can be critically tied. In fact, in the case of the Siemens suite programs (regardless of which fault localization technique is used), only 26 of the 129 faulty versions (20.16%) are free of critical ties. Similarly, critical ties are observed in case of the gzip, grep, and make programs as well, even though the scale does not seem as significant as in the case of the Siemens suite. This illustrates that critical ties are not a rare occurrence, and when they do occur, a significant portion of the code might be critically tied. Thus, this justifies the need for tie-breaking strategies.

3. The Tie-Breaking Strategies

Four tie-breaking strategies are considered in our experiments that can be classified as (1) statement order based (SOS), (2) confidence based (two strategies — CS1 and CS2), and (3) data-dependency based (DDS). Fault localization techniques already sort statements in descending order of their suspiciousness, and the tie-breaking strategies are only used to resolve ties. For the purposes of evaluating a tie-breaking strategy we only focus on critical ties. Let us assume CT is a critical tie consisting of k statements, and $\Omega(st)$ — where statement $st \in CT$ — represents a metric based on which ties among these k statements can be resolved such that a statement with a larger Ω value will be examined before a statement with a smaller Ω value.

Statement order based strategy (SOS): This strategy calls for tied statements to be internally sorted in the order in which they appear in the code: their statement number. In this scenario,

$$\Omega(st) = -(\text{the statement number of } st).$$

This strategy assumes that when programmers debug, they intuitively follow the order in which the program may appear (say if source file is opened in an Integrated Development Environment or a plain text editor). Take special note of the minus sign in the Ω function for the SOS strategy, which has an important purpose. It is necessary because statements which appear first are assigned lower statement numbers; and without the minus sign, they would be examined only after those with higher statement numbers, which appear later. Thus, the minus sign in the Ω function of the SOS strategy allows us to follow our convention that statements with larger Ω values are examined before those with smaller Ω values, while also ensuring that if two or more statements are tied, then statements that appear first are also examined first.

Confidence based strategy (CS): We propose a strategy based on the intuition that statement suspiciousness should be more dependent on the information extracted from failed test cases than successful ones. Thus, regarding the suspiciousness computation of a statement — the contribution of the failed test

cases that execute it should be greater than the contribution of the successful test cases that execute it [17]. Thus, the metric is named so, as it is designed to measure the degree of confidence in a given suspiciousness value, implying that when two or more statements are assigned the same suspiciousness value, the suspiciousness assigned to statements with a higher confidence is more reliable and the corresponding statements are more likely to contain faults. We use a confidence metric for CS1:

$$confidence(s) = \frac{failed(s)}{totalfailed} - \frac{successful(s)}{totalsuccessful}. \tag{4}$$

A similar approach is proposed in [22], which the authors refer to as the 'confidence' of a statement. The metric is given as follows, and the strategy that utilizes it is referred to as CS2.

$$confidence(s) = \max\left(\frac{failed(s)}{totalfailed}, \frac{successful(s)}{totalsuccessful}\right). \tag{5}$$

Regardless of whether CS1 or CS2 is used to break-ties, we have $\Omega(st) = confidence(st)$: the tied statements with higher confidence values are examined before those with lesser values.

Data dependency based strategy (DDS): The use of data dependency analysis has been proposed for the purposes of fault localization in [19]. Let a data dependency relation Δ between two statements s_i and s_j be defined such that $s_i \,\Delta\, s_j$ if and only if s_i defines some variable that is used in s_j, or s_j defines some variable used in s_i.[e] For a given statement st in the critical tie CT, its Ω value is defined as the suspiciousness value of the most suspicious statement that is not in the critical tie, but is data dependent on st.

$$\Omega(st) = \max(susp(s) | \forall s(s \notin CT) \wedge (s \Delta st)) \tag{6}$$

The rationale behind this strategy is that even though a fault may not be in a statement st that has a high suspiciousness value, it is very likely that the fault has something to do with a statement that is directly data dependent on st.

4. Evaluation of the Tie-Breaking Strategies

In this section we first present the metric used for evaluating the effectiveness of a tie-breaking strategy, followed by the results of our case studies which show how effective the strategies proposed in Sec. 3 are at breaking the critical ties.

4.1. *Evaluation metric*

To evaluate the effectiveness of tie-breaking strategies, we need to measure how well they reduce the size of critical ties. To do this we define a quantity *Tie-Reduction*

[e]Note that this relation is symmetric, i.e., $s_i \,\Delta\, s_j \iff s_j \,\Delta\, s_i$, but not necessarily transitive.

which measures the extent to which critical ties have been broken. For any tie-breaking strategy, its corresponding *Tie-Reduction* is computed as:

$$\textit{Tie-Reduction} = \left(1 - \frac{\text{size of a critical tie after applying the strategy}}{\text{size of a critical tie without applying the strategy}}\right) \times 100\%. \tag{7}$$

The primary motivation of using a tie-breaking strategy is to provide programmers with an idea of which statements to examine first among those that are tied, and to have a better idea of the expected effectiveness of a fault localization technique (by reducing, if not removing, the gap between the best and worst effectiveness — referring to Sec. 1). However, it is important to recognize that the breaking of ties can alter the effectiveness of a fault localization technique, and in order for a tie-breaking strategy to be useful, it should not affect the fault localization effectiveness adversely. Thus, this becomes a fundamental requirement of any proposed tie-breaking strategy, and so along with measuring the *Tie-Reduction* of tie-breaking strategies, we also define a quantity *Impact*, to evaluate the change in fault localization effectiveness caused by the use of tie-breaking strategies. For the purposes of this paper, fault localization effectiveness is measured in terms of the average number of statements that need to be examined to find the first faulty statement: the average effectiveness computed as the mean of the best and worst effectiveness; for a discussion on other metrics, please refer to Sec. 7. Thus, for any faulty program, the *Impact* of a tie-breaking strategy can be computed as

$$\textit{Impact} = \frac{|S_{before} - S_{after}|}{\text{Total number of statements}} \times 100\% \tag{8}$$

where S_{before} and S_{after} refer to the average number of statements that need to be examined before and after using a tie-breaking strategy, respectively. A good tie-breaking strategy should strive to at least maintain the same average fault localization effectiveness (a low *Impact* if not zero), while resulting in a high *Tie-Reduction* (considerably reducing the size of the critical tie).

4.2. *Results*

Table 2 presents the cumulative number of critically tied statements (CCTS) across all faulty versions studied, and the corresponding *Tie-Reduction* (see Eq. (7)), when each of the strategies is used in conjunction with the three fault localization techniques described in Sec. 2.2, with respect to each program. In Table 2 we annotate the *Tie-Reduction* as '*TR*'. For example, from the first row of the table, we observe that 858 statements are critically tied without the use of any tie-breaking strategy when using Tarantula on the programs of the Siemens suite. However, when we attempt to break ties we see that SOS breaks all the critical ties, thereby achieving a *Tie-Reduction* of 100%; the CS1 and CS2 strategies are not able to

Table 2. Evaluation of Tie-breaking strategies in terms of Tie-Reduction.

		Strategies								
		None	SOS		CS1		CS2		DDS	
Program	Techniques	CCTS	CCTS	TR	CCTS	TR	CCTS	TR	CCTS	TR
Siemens	Tarantula	858	0	100%	858	0%	858	0%	773	9.91%
	Ochiai	896	0	100%	896	0%	896	0%	810	9.60%
	Heuristic III	896	0	100%	896	0%	896	0%	818	8.71%
gzip	Tarantula	1922	0	100%	1777	7.54%	1777	7.54%	1565	18.57%
	Ochiai	1777	0	100%	1777	0%	1777	0%	1438	19.08%
	Heuristic III	1778	0	100%	1777	≈0%	1777	≈0%	1435	19.29%
grep	Tarantula	2019	0	100%	1733	14.17%	1733	14.17%	1607	20.41%
	Ochiai	1733	0	100%	1733	0%	1733	0%	1475	14.89%
	Heuristic III	1733	0	100%	1733	0%	1733	0%	1483	14.43%
make	Tarantula	6578	0	100%	5739	12.75%	5739	12.75%	5861	10.90%
	Ochiai	5739	0	100%	5739	0%	5739	0%	5165	10.00%
	Heuristic III	5739	0	100%	5739	0%	5739	0%	5183	9.69%

break any critical ties, which corresponds to a *Tie-Reduction* of 0%; and the DDS strategy is able to reduce critical ties by almost 9.91%. Additionally we observe that:

(1) The SOS strategy achieves a *Tie-Reduction* of 100% for every case in that all ties are broken. This is because in the case of SOS, statement numbers are used to break critical ties; and as long as each statement number is unique, there will be no ties.
(2) CS1 and CS2 are not effective in most cases. Of the 207 faulty versions used in our study, 28 are free of critical ties. 165 of the remaining versions have critical ties (irrespective of fault localization technique) as the tied statements are executed by the same number of failed and successful test cases. Since the total number of failed and successful test cases is fixed with respect to each statement, it is clear from Eqs. (4) and (5) that these statements have the same Ω value. As a result, the ties cannot be broken by CS1 and CS2 which significantly reduces their tie-breaking effectiveness. Next, we explain why it is possible that CS1 and CS2 may work for the remaining 14 ($207 - 28 - 165 = 14$) faulty versions when Tarantula is used.

For the last 14 versions, 11 of them have critical ties that only contain statements not executed by any successful test cases. In such a scenario, the Ochiai and Heuristic III techniques can still distinguish between such statements while Tarantula cannot. To better understand this, consider two statements s_1 and s_2 which are executed by 2 and 3 failed test cases, respectively, but not by any successful test cases ($\text{successful}(s_1) = \text{successful}(s_2) = 0$). Also assume the total number of failed test cases (*totalfailed*) is 3, and that the total number of successful test cases (*totalsuccessful*) is 5. If Ochiai is used, the suspiciousness assigned to statement s_1 is

about 0.8165 and that assigned to statement s_2 is 1, and so, Ochiai will not tie these statements. The same is true of Heuristic III. But as per Eq. 1, the suspiciousness computation for statements s_1 and s_2 will both evaluate to 1 according to Tarantula, which means that if Tarantula is used, they will be tied. However, to continue our example, if CS1 and CS2 are now applied then statements s_1 and s_2 can be distinguished from one another (by virtue of the number of failed test cases that execute the statements). Thus, CS1 and CS2 are especially effective at breaking this kind of tie which is specific to Tarantula. It explains why these strategies show positive results on the 11 faulty versions under discussion. This does not mean that the Ochiai and Heuristic III techniques are free from critical ties in the case of these faulty versions, but rather that statements may not be critically tied for the same reasons as with the Tarantula technique.

That brings us to the discussion on why CS1 and CS2 seem to share the same effectiveness. Recall that CS1 and CS2 seem most effective only in the case of Tarantula on 14 (out of 207) faulty versions. For 11 of these faulty versions, (as discussed above) Tarantula results in critical ties, where some statements are not executed by any successful test cases. In this situation, for both CS1 and CS2 the Ω value shall be equal to the fraction $failed(s)/totalfailed$ as per Eqs. (4) and (5), which in turn implies that the two strategies are one and the same for these 11 cases, and thereby equally as effective. Coincidentally, CS1 and CS2 also have the same results on the remaining three $(14 - 11 = 3)$ versions. CS1 and CS2 are further discussed in Sec. 5.

(3) In terms of *Tie-Reduction*, the DDS strategy is able to reduce critical ties by at least 8.71%, and even achieves a maximum reduction of 20.41% (Tarantula on grep). DDS also seems to be more stable in the sense that regardless of fault localization technique, or subject program, we always observe some reduction in critical ties by applying the DDS strategy.

Now recall that a good tie-breaking strategy should maintain the same fault localization effectiveness as prior to its application; and to evaluate this, we defined the quantity *Impact* as per Eq. (8) in Sec. 4.1. Since our case study encompasses multiple programs, and furthermore, multiple faulty versions for each of these programs, and because we have three different fault localization techniques and also four different tie-breaking strategies; for ease of representation, we present the *Impact* in terms of an average. Table 3 presents the average *Impact* (which is

Table 3. The average *Impact* on the effectiveness of fault localization techniques by tie-breaking strategies.

	SOS	CS1	CS2	DDS
Tarantula	2.36%	0.08%	0.08%	0.36%
Ochiai	2.38%	0%	0%	0.35%
Heuristic III	2.38%	≈0%	≈0%	0.35%

averaged across all 207 faulty versions under study) using each of the four tie-breaking strategies applied in conjunction with each of the three fault localization techniques. We observe that in the worst case SOS perturbs the effectiveness by almost 2.38%. In fact, this is an improvement on the fault localization effectiveness, which warrants further discussion on SOS, and therefore, this issue is picked up again in Sec. 5.3, CS1, CS2 and DDS have just a slight *Impact* on the fault localization effectiveness: only about 0.35% or 0.36% in the worst case.

The data from the case studies allows us to have a considerable degree of confidence in the claim that the proposed strategies can break ties to a significant extent, yet do so without adversely affecting the effectiveness of the techniques.

5. Discussion

In this section, we discuss several issues that are related to the work that is presented in this paper: evaluating the tie-breaking strategies with respect to all ties and not just critical ties; presenting further details on the SOS, CS1 and CS2 strategies; the proposal of a new metric for better evaluating fault localization techniques, and a discussion on a new slicing-based tie-breaking strategy.

5.1. *Tie-breaking strategies: Critical ties versus all ties*

As per the discussion in Sec. 2, to evaluate the tie-breaking strategies, essentially we only need to focus on the impact that they have on critical ties. But clearly, when the tie-breaking strategies are applied, they can work for all ties including the ones that are non-critical. In this section, we present data on how the tie-breaking strategies work with respect to all ties. To facilitate discussion, the programs of the Siemens suite are utilized as subject programs, and the results of this portion of the study are presented in Table 4, across all the programs and faulty versions of the Siemens suite collectively. Taking the example of the "Tarantula" data in row 1, we observe that there are a total of 14255 tied statements (out of 16507 executable statements in all) without the use of any tie-breaking strategy, while there are 13591 (a reduction of 4.66%) using CS1, 13591 (a reduction of 4.66%) using CS2, and

Table 4. Evaluation of tie-breaking strategies for all ties in the Siemens suite.

		Strategies					
		CS1		CS2		DDS	
Techniques	Total number of tied statements	Number of tied statements	Reduction	Number of tied statements	Reduction	Number of tied statements	Reduction
Tarantula	14255	13591	4.66%	13591	4.66%	12560	11.89%
Ochiai	14255	13591	4.66%	13591	4.66%	12588	11.69%
Heuristic III	13592	13591	≈0%	13591	≈0%	11839	12.90%

12560 (a reduction of 11.89%) using DDS. In the table we exclude the data for SOS, because as we pointed out in Sec. 4.2, SOS breaks all ties and therefore the number of tied statements will always be zero.

Although both Table 2 and Table 4 focus on the reduction of ties, there is a subtle difference between them in terms of how the corresponding reduction is measured. Table 2 only focuses on critical ties, and is indifferent to whether the tied statements that have been broken would form another non-critical tie. This can happen because two or more statements might be broken off from a critical tie but still be tied with respect to each other. As opposed to Table 2, Table 4 takes all the ties into consideration and when measuring reduction, only counts the tied statements that are broken and does not then go on to form other ties, non-critical as they may be. As per our discussion in Sec. 2, every statement in a non-critical tie which has a higher suspiciousness assigned to it than that of the statements in a critical tie, needs to be examined regardless of the internal order of examination; while non-critical ties with a lesser suspiciousness have no reason to be examined, as we would stop after encountering the faulty statement in the critical tie. However, even though the breaking of non-critical ties may not be as useful as the breaking of critical ones, the data presented in this section still validates the proposed strategies in terms of their tie-breaking abilities.

5.2. *Combining tie-breaking strategies together*

Since the proposed strategies try to break ties from different perspectives (SOS tries to break ties based on the statement number assigned to statements, CS1 and CS2 try to break ties using the execution information and DDS takes data flow information into consideration), it follows that multiple strategies may also be applied in conjunction with one another. In this section, we illustrate how strategies may be used together, and for the purposes of illustration choose the DDS and SOS strategies (because they are completely independent of one another). This is only one possible combination of many, and indeed more than two strategies may also be used together. In this example, DDS is used as the first tie-breaking strategy and SOS as the secondary: first DDS is used to break the critical tie, and then SOS is used to break the remaining critical tie, if necessary. We choose gzip (on which DDS performs reasonably well based on Table 2, but does not break critical ties completely) as our subject program.

Table 5 compares the tie-breaking abilities of the SOS, DDS, and combined DDS and SOS (DDS + SOS) strategies on the gzip program. For ease of reference, the data corresponding to when no tie-breaking strategy is used is also provided as per the column labeled 'None'. We evaluate with respect to two quantities — the "Cumulative Critically Tied Statements (annotated as *CCTS* in the table)" and the "Cumulative Average Effectiveness (annotated as *CAE* in the table)". For each strategy; the first column shows the number of critically tied statements after using the strategy, and the second column shows the percentage improvement as a result

Table 5. The comparison between SOS, DDS and DDS + SOS on the gzip program.

		Strategies						
			SOS		DDS		DDS + SOS	
Technique	Evaluation Quantity	None	SOS	Improvement	DDS	Improvement	DDS + SOS	Improvement
Tarantula	CCTS	1922	0	100%	1565	18.57%	0	100%
	CAE	4071	4060	0.27%	4037.5	0.82%	4010	1.50%
Ochiai	CCTS	1777	0	100%	1438	19.08%	0	100%
	CAE	2158.5	2155	≈0%	2106	2.43%	2083	3.50%
Heuristic III	CCTS	1778	0	100%	1435	19.29%	0	100%
	CAE	2424	2420	≈0%	2373.5	2.08%	2352	2.97%

of using the strategy. A positive percentage in the improvement column indicates that the total number of critically tied statements is reduced or the cumulative average effectiveness is improved. From Table 5, we observe that the combination of DDS and SOS is able to break all the ties (which is one of the benefits of using SOS) without bringing down the effectiveness (in fact it improves the effectiveness by at least 1.5% which is more than the use of either the SOS or the DDS strategy alone).

Finally, the purpose of this section is only to discuss a potential way in which strategies may be used together, and to further validate the effectiveness of using multiple strategies simultaneously, more cases studies need to be performed which is part of our future work.

5.3. *Why does SOS seem to improve the effectiveness of fault localization techniques?*

Based on the data presented in this paper, it seems that not just does SOS consistently alter the effectiveness of a fault localization technique, but strangely enough, it also slightly improves it. Under normal circumstances there should be no reason for this, as the statement number assigned (by, say a text editor) is not biased towards any statement, and indeed a text editor does not even know which statement is faulty. Thus, at the risk of conjecturing, our observations are probably linked to external factors such as the way in which faults have been injected into the subject programs.

To better understand what we mean, let us revisit the SOS strategy and identify the circumstances under which the average effectiveness might improve by its application. Since the SOS strategy will only internally sort the statements in a tie, based on their statement number, if the effectiveness increases then it is likely because the faulty statement in a critical tie is being assigned a relatively lower statement number (by IDEs and text-editors, not by us) than other statements in a critical tie. Since such a statement number is assigned based on the order in which

statements *appear*, it follows that the faulty statements in our studies appear early on with respect to the other statements in a critical tie. But why would this be the case?

Let us keep in mind the question posed above, but deviate for just a moment. The most common situation, under which the fault localization techniques discussed in this paper would assign the same suspiciousness to a set of statements (resulting in a tie), is if each of them was executed by the same number of failed and successful test cases. This in turn is definitely going to be the case for a set of statements that are executed as an atomic block (such that if one statement in the block is executed, then all will be executed). To continue this discussion let us assume that a fault injected in a program (i.e., the faulty statement) is part of one such hypothetical block.

We believe that when someone, regardless of their relative experience, is presented with a set of statements as part of one block in which to inject a fault, the natural inclination is to inject a fault in the first available location, starting from the order of appearance. Given n statements that appear in the order 1 to n, the tendency is probably to inject a fault in the first statement, then move on if necessary to the second statement and so on, until ultimately a fault may be injected in the *nth* statement if necessary. If only one fault is to be injected, then it will most likely be at the locations/statements that appear early on, if not the earliest.

This is most likely the reason that SOS is able to increase the effectiveness of the fault localization techniques. Among critically tied statements that involve the faulty statement as part of an atomic block, the fault has typically been injected early on, in order of appearance, in the block, and there have not always been faults injected in the latter part of the blocks. The net effect has been for the faulty statement to be assigned a lower statement number than the non-faulty statements, simply because it appears first, thereby increasing the fault localization effectiveness once SOS is applied. This same phenomenon is true even if the faults are injected using mutation operators, as it is not so much a question of what sort of fault has been injected, as much as of where it has been injected, and this is most typically the first available location in order of appearance.

This is our explanation as to why the SOS strategy seems to alter, and in fact improve, the fault localization effectiveness. We posit that if faults are injected in a purely random manner without any bias to location, then the SOS strategy will probably not result in an improved effectiveness. An exploration and validation of this conjecture is one of our future works.

5.4 *The Circumstances under which CS1 and CS2 work, and why their results are generally similar*

Based on the data in Secs. 4.2 and 5.1, we find that the CS1 and CS2 strategies do not work very well in our case studies. Then under what circumstances will they be of use?

Table 6. Scenarios where CS1 and CS2 may be effective at breaking ties.

Techniques	Scenarios
Tarantula	Scenario 1: Both *successful* (s_1) and *successful* (s_2) are zero, but *failed* (s_1) and *failed* (s_2) are different.
	Scenario 2: Both *failed* (s_1) and *failed* (s_2) are zero, but *successful* (s_1) and *successful* (s_2) are different.
	Scenario 3: *failed* (s_1)/*failed*(s_2) = *successful* (s_1)/*successful* (s_2), but *failed* (s_1) and *failed* (s_2) are different
Ochiai	Same as Scenario 2.
Heuristic III	Scenario 4: $failed(s_1) = 1$, $failed(s_2) = 2$, $successful(s_1) > 0$, successful $(s_2) >= 1$, *and* $successful(s_1) = successful(s_2) - 1$

The three fault localization techniques employed in our study compute the suspiciousness of each statement by using information such as the number of failed and successful test cases that execute the statement, and the total number of failed and successful test cases. The CS1 and CS2 strategies also use the same information to break ties. Note that for a given faulty version, *totalfailed* and *totalsuccessful* are the same for every statement. Hence, if statements are tied because they have the same number of failed and successful test cases, then CS1 and CS2 will not work. Stated differently, CS1 and CS2 can break ties under the condition that statements have the same suspiciousness but are executed by different number of failed or successful test cases. This condition can be observed under different scenarios some of which are listed in Table 6, with respect to each fault localization technique. For example, in Scenarios 1, 2 and 3, CS1 and CS2 are able to break ties between statements s_1 and s_2, where Tarantula assigns the same suspiciousness value to them.

Recall our discussion in Sec. 4.2 where we discussed the similarity in results obtained by using CS1 and CS2. We now extend that discussion by further analysis of the data in Tables 2 and 4 and only focus on the cases where CS1 and CS2 work. As also discussed in Sec. 4.2, CS1 and CS2 co-incidentally produce the same results for the three versions studied. In fact these three faulty versions are all cases of Scenario 3 (as per Table 6). Although CS1 and CS2 are able to break ties under this scenario, and furthermore the Ω values (computed by Eqs. (4) and (5)) for CS1 and CS2 are different, the examination order for the tied statements (based on their Ω value) does not differ for CS1 and CS2. This implies that CS1 and CS2 produce the same results.

Regarding the results with respect to Tarantula and Ochiai in Table 4 (all ties and not just critical ties), CS1 and CS2 are able to break ties in 91 versions out of a total of 129 faulty versions of the Siemens suite. These 91 faulty versions are all cases of Scenario 2 (refer to Table 6) and thus, the Ω values for both CS1 and CS2 will be equal to the fraction $successful(s)/totalsuccessful$, which in turn means that CS1 and CS2 have the same results. Finally, with respect to the results on Heuristic III in Table 4, CS1 and CS2 can only break ties for one version, which

is version 25 of the *replace* program. In this case also, similar to the three faulty versions discussed with respect to Tarantula above, CS1 and CS2 coincidentally lead to the same results. In summary, CS1 and CS2 provide the same results based on our case studies, though this may not always be the case.

5.5 *A new metric for evaluating fault localization techniques*

As discussed extensively in this paper, critical ties lead to two different levels of effectiveness — the best and the worst, and therefore the expected effectiveness can be expressed as the average of the best and the worst effectiveness. However, let us consider a hypothetical faulty program on which a fault localization technique α requires the examination of ω statements less than another technique β in the best case, but ω statements more in the worst case. Clearly, techniques α and β result in the same average effectiveness, but are they really as effective equally?

By virtue of investigating critical ties in this paper, we also postulate that given the above scenario, technique β is more effective than technique α as the number of statements that are critically tied is less than in the case of α. With this in mind, we would like to discuss a new evaluation metric — *Critical Score* (*CScore* for short) — which takes into account the number of critically tied statements based on the use of a particular fault localization technique. Formally we have,

$$CScore = \left(\frac{\text{number of critically tied statements}}{\text{total number of statements}}\right) \times 100\%. \tag{9}$$

For a fault localization technique, a low *CScore* is desirable as it implies a fewer number of statements are critically tied. However, as an evaluation metric, it is better to make use of the *CScore* in conjunction with other metrics that evaluate fault localization effectiveness, such as the total number of statements that need to be examined to locate a fault. We believe that the *CScore* is secondary to fault localization effectiveness and only comes into play when two techniques share the same or comparable effectiveness. However, as revealed by this paper, ties represent a significant problem with regards to fault localization, and therefore, it is foreseeable that the *CScore* might soon be an important consideration. We intend to evaluate how much weight or importance to assign the *CScore* metric with respect to traditional effectiveness metrics as part of our future work.

5.6 *A novel slicing-based approach to breaking ties*

As mentioned in Secs. 4.2 and 6, most of the tied statements have not only the same suspiciousness, but also the same number of successful and failed test cases that execute them. This hints at the fact that using execution trace information, such as in CS1 and CS2, is not good enough for tie breaking. In this paper, we discuss a strategy based on the data dependency (the DDS strategy). In this section, we would like to introduce another possible strategy for tie-breaking, hoping to guide

Table 7. An example for slicing based tie-breaking strategy.

Tested program	Test cases t_1 (3, −2, 1)	t_2 (3, −1, 1)	t_3 (2,3,1)	t_4 (−3, 2, 1)	susp
test (int a, int x, int y) {					
int b, z;					
1: $a = a + x$;	1	1	1	1	0.5
2: $b = y$ // fault: correct: $b = x$	1	1	1	1	0.5
3: if ($a > 1$)	1	1	1	1	0.5
4: $b = a - 4$;	0	1	1	0	0
5: if ($b > 0$)	1	1	1	1	0.5
6: $z = x + y$;	1	0	1	0	0.5
else					
7: $z = x + y + 1$;	0	1	0	1	0.5
8: output(z);	1	1	1	1	0.5
}					
Result (Success — P/Failure — F)	F	P	P	F	

readers who are interested in the area, and stimulate further research. We believe that slicing techniques (for our purposes — *dynamic* slicing techniques) can be used to break ties, and we refer to this category of strategies as *slicing-based*.

An illustrative example is given in Table 7, where the example program consists of eight executable statements, out of which statement 2 is faulty. To the right of the code is a test suite containing four test cases, and for each test case, its input is shown at the top of its column, while its coverage is shown within the column. A '1' in the coverage column means the corresponding statement is executed and a '0' means that it is not. The execution result is shown at the bottom of the column, where 'P' means the corresponding test case succeeded and 'F' means it failed. Tarantula is used to compute the suspiciousness which is in 'susp' column of each statement.

Analyzing the suspiciousness reveals that there are seven statements (statements 1, 2, 3, 5, 6, 7, and 8) that are critically tied. To use dynamic slicing for tie-breaking, we first construct the dynamic slice with respect to each failed test case, and then consider the intersection of these slices, which in the case of the example in Table 7 is {2, 5, 8}. Within the critical tie, we give higher examination priority to the statements in the intersection. In our example, {2, 5, 8} would be examined before {1, 3, 6, 7}. Since {2, 5, 8} does indeed contain the faulty statement, we can say that dynamic slicing has reduced the size of the critical tie from seven to three statements. The rationale for using dynamic slicing is that it can extract the statements that influence the computation of faulty output values, and many case studies [25] have shown that some types of slicing can locate faulty statements effectively. Therefore, slicing can also be used to reduce the size of the critical tie effectively. This is only to serve as an example to readers who would like to work in this area. More case studies need to be conducted to thoroughly evaluate this strategy.

6. Threats to Validity

There are several identified threats to validity which include, but may not be limited to, the following. A threat to external validity is our choice of fault localization techniques and subject programs which may limit our ability to generalize our results. To expose the significance of ties and to evaluate the tie-breaking strategies proposed herein, three fault localization techniques — Tarantula, Ochiai, and Heuristic III — were evaluated across four sets of subject programs — the programs of the Siemens suite and the grep, gzip and make programs (10 programs in total corresponding to 207 faulty versions in total). While we cannot claim that our results apply everywhere, this extensive combination of techniques and programs does allow us to have confidence in the validity of our results.

A threat to the internal validity is the way in which the faults may have been inserted in order to create the faulty versions that have been used for our experiments. Even though the results in Sec. 4.2 indicate that the statement order based strategy (SOS) can potentially improve the effectiveness of a fault localization technique, we are unable to attribute a reason as to why (but may only conjecture as has been done in Sec. 5.3). It most likely has something to do with the faults, or the way they were seeded. A threat to the construct validity is the metric used to evaluate fault localization effectiveness, i.e., the total number of statements that need to be examined to locate a fault. However, such a metric has also been used in studies such as [3, 17].

Another threat to the validity is that the subject programs used for our experiments have all been single-fault programs (each faulty version has exact one fault in it). However, tie-breaking strategies may easily be applied to programs with multiple faults as well, via the procedure described in [8] where *fault-focused* clusters are constructed such that failed tests in each cluster are associated with the same fault. The successful test cases are then combined with the fault-focused clusters to produce specialized test suites, whose information can be fed to a fault localization technique to locate the fault associated with that cluster. Once such fault-focused clusters are constructed, regardless of which fault localization technique is used, statements may still be tied for the same suspiciousness, and therefore, the tie-breaking strategies are still of great value.

7. Related Work

There are many fault localization techniques, in addition to the ones discussed by us, that have been proposed over the recent past, and while this paper is directly related to fault localization, we cannot hope to cover them all. For readers interested in finding out more about fault localization and the current state of the art fault localization techniques, we refer them to [16]. And there are indeed some other strategies to improve the fault localization techniques. Masri *et al.* [5] and Wang *et al.* [15] proposed some strategies to clean or trim coincidental correctness so as to improve the effectiveness of fault localization techniques.

Debroy *et al.* [3] proposed a grouping-based strategy which groups program components based on the number of failed tests that execute that component and ranks the group that contains components that have been executed by more failed tests; thus the statements are examined firstly based on the their group order and secondarily based on their suspiciousness, which is computed by fault localization techniques. It is possible that the grouping-strategy may be modified to work as a tie-breaking strategy as well. One way to do this would be to use a fault localization technique, and then break ties with respect to the number of failed test cases that execute the statements. However, the usefulness of this modified grouping strategy has yet to be evaluated. With the exception of the confidence metric (used in strategy CS2) proposed by Jones *et al.* in [22], we are unaware of any other research work that has dealt explicitly with the problem of resolving ties (especially critical ties) in the context of fault localization.

In this paper we have discussed a new metric (to be used in conjunction with other metrics) to evaluate the effectiveness of a fault localization technique. Aside from this metric, we have evaluated fault localization effectiveness in terms of the total number of executable statements that must be examined in order to locate all faults under study. Other metrics also exist that evaluate fault localization effectiveness using various means. Renieris *et al.* [11] suggested the assignment of a score to each faulty version of a subject program, which is defined as the percentage of the program that need not be examined to find a faulty statement in the program or a faulty node in the corresponding program dependence graph (PDG). Cleve and Zeller also adopted the same score (effectiveness measure) in [2]. To make their computations comparable to those based on the PDG, the authors of Tarantula [7] consider only executable statements but essentially use the same score. Wong *et al.* [17] define an *EXAM* score which expresses the effectiveness of a fault localization technique in terms of the percentage of code that needs to be examined, as opposed to the percentage of code that need not. A similar modification is made by Liu *et al.* in [9] except that their score is again based on the PDG.

8. Conclusions

Fault localization techniques strive to assign suspiciousness (likelihood of being faulty) values to program components, such that these components can then be examined by programmers in decreasing order of their suspiciousness during debugging. However, statements might be tied for the same suspiciousness, and thus programmers must use their own intuition or judgment to decide which components to examine first. Furthermore, the greater the number of ties that involve faulty statements, the harder it is to precisely estimate at what point during the examination the faulty statement will be encountered. In this paper we systematically analyze the problems associated with ties involving faulty statements and reveal that the problem is quite frequent and is not limited to any particular fault localization technique or subject program. To alleviate the problem, four

tie-breaking strategies are considered and evaluated via case studies. Results indicate that some of the strategies can indeed reduce ties without having an adverse impact on fault localization effectiveness. More specifically, the SOS strategy is able to break all ties, regardless of whether they are critical or non-critical; CS1 and CS2 strategies are also able to break the critical ties to some extent, though they may only work under specific scenarios, and the DDS strategy is able to reduce critical ties by at least 8.71%, up to 20.41%. We also discuss a slicing-based strategy that holds potential in terms of its ability to break ties and present a metric, *CScore*, that takes ties into account when evaluating fault localization techniques, which is to be used in conjunction with existing evaluation metrics. Our future work includes evaluating other tie-breaking strategies that may be based on a variety of different intuitions and assessing the usefulness of the *CScore* metric, as opposed to traditional metrics, on a large scale. We also wish to extend our studies to programs with multiple faults in them.

References

1. R. Abreu, P. Zoeteweij, R. Golsteijn and A. J. C. van Gemund, A practical evaluation of spectrum-based fault localization, *Journal of Systems and Software* **82**(11) (2009) 1780−1792.
2. H. Cleve and A. Zeller, Locating causes of program failures, in *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, Missouri, May 2005, pp. 342−351.
3. V. Debroy, W. Eric Wong, X. Xu and B. Choi, A grouping-based strategy to improve the effectiveness of fault localization techniques, in *Proceedings of the 10th International Conference on Quality Software*, Zhangjiajie, China, July 2010, pp. 13−22.
4. http://sir.unl.edu/portal/index.html (Software-artifact Infrastructure Repository).
5. W. Masri and R. A. Assi, Cleansing test suites from coincidental correctness to enhance fault-localization, in *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*, Paris, France, April 2010, pp. 165−174.
6. D. Jeffrey, N. Gupta and R. Gupta, Fault localization using value replacement, in *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA)*, Seattle, Washington, July 2008, pp. 167−178.
7. J. A. Jones and M. J. Harrold, Empirical evaluation of the Tarantula automatic fault-localization technique, in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, California, USA, November 2005, pp. 273−283.
8. J. A. Jones, J. Bowring and M. J. Harrold, Debugging in parallel, in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, London, UK, July 2007, pp. 16−26.
9. C. Liu, L. Fei, X. Yan, J. Han and S. P. Midkiff, Statistical debugging: a hypothesis testing-based approach, *IEEE Transactions on Software Engineering*, **32**(10) (2006) 831−848.
10. C. Liu and J. Han, Failure proximity: A fault localization-based approach, in *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Portland, Oregon, November 2006, pp. 286−295.

11. M. Renieris and S. P. Reiss, Fault localization with nearest neighbor queries, in *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, Montreal, Canada, October 2003, pp. 30−39.
12. R. Santelices, J. A. Jones, Y. Yu and M. Jean Harrold, Lightweight fault-localization using multiple coverage types, in *Proceedings of the 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009, pp. 56−66.
13. The Siemens Suite, http://www-static.cc.gatech.edu/aristotle/Tools/subjects/, January 2007.
14. I. Vessey, Expertise in debugging computer programs, *International Journal of Man-Machine Studies: A Process Analysis* **23**(5) (1985) 459−494.
15. X. Wang, S. C. Cheung, W. K. Chan and Z. Zhang, Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization, in *Proceedings of the 31st International Conference on Software Engineering*, Vancouver, Canada, May 2009, pp. 45−55.
16. W. E. Wong and V. Debroy, A survey on software fault localization, Technical Report UTDCS-45-09, Department of Computer Science, University of Texas at Dallas, November 2009.
17. W. E. Wong, V. Debroy and B. Choi, A family of code coverage-based heuristics for effective fault localization, *Journal of Systems and Software* **83**(2) (2010) 188−208.
18. W. E. Wong and Y. Qi, BP neural network-based effective fault localization, *International Journal of Software Engineering and Knowledge Engineering*, **19**(4) (2009) 573−597.
19. W. E. Wong and Y. Qi, Effective program debugging based on execution slices and inter-block data dependency, *Journal of Systems and Software*, **79**(7) (2006) 891−903.
20. W. E. Wong, Y. Shi, Y. Qi and R. Golden, Using an RBF neural network to locate program bugs, in *Proceedings of the 19th IEEE International Symposium on Software Reliability Engineering*, Seattle, USA, November 2008, pp. 27−38.
21. W. E. Wong, T. Wei, Y. Qi and L. Zhao, A crosstab-based statistical method for effective fault localization, in *Proceedings of the First International Conference on Software Testing, Verification and Validation*, Lillehammer, Norway, April 2008, pp. 42−51.
22. Y. Yu, J. A. Jones and M. J. Harrold, An empirical study of the effects of test-suite reduction on fault localization, in *Proceedings of the International Conference on Software Engineering*, Leipzig, Germany, May 2008, pp. 201−210.
23. χSuds User's Manual, Telcordia Technologies, (1998).
24. X. Zhang, N. Gupta and R. Gupta, Locating faults through automated predicate switching, in *Proceeding of the 28th International Conference on Software Engineering*, Shanghai, China, May 2006, pp. 272−281.
25. X. Zhang, H. He, N. Gupta and R. Gupta, Experimental evaluation of using dynamic slices for fault location, in *Proceedings of the sixth International Symposium on Automated Analysis-driven Debugging*, Monterey, California, September 2005, pp. 33−42.
26. A. X. Zheng, M. I. Jordan, B. Liblit, M. Naik and A. Aiken, Statistical debugging: simultaneous identification of multiple bugs, in *Proceedings of the International Conference on Machine Learning*, Pittsburgh, Pennsylvania, June 2006, pp. 1105−1112.