SOFTWARE – PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2011)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.1146

A consensus-based strategy to improve the quality of fault localization

Vidroha Debroy and W. Eric Wong*,†

Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75080-3021, USA

SUMMARY

A vast number of software fault localization techniques have been proposed recently with the growing realization that manual debugging is time-consuming, tedious, and error-prone, and fault localization is one of the most expensive debugging activities. Although some techniques perform better than others on a large number of data sets, they do not do so on all data sets and therefore, the actual quality of fault localization can vary considerably by using just one technique. This paper proposes the use of a consensus-based strategy that combines the results of multiple fault localization techniques to consistently provide high quality performance, irrespective of data set. Empirical evidence based on case studies conducted on six sets of programs (seven programs of the Siemens suite, and the gzip, grep, make, space, and Ant programs) and three different fault localization techniques (Tarantula, Ochiai, and H3) suggests that the consensus-based strategy holds merit and generally provides close to the best, if not the best, results. Empirically, we show that this is true of both single-fault and multifault programs. Additionally, the consensus-based strategy makes use of techniques that all operate on the same set of input data, minimizing the overhead. It is also simple to include or exclude techniques from consensus, making it an easily extensible or tractable strategy. Copyright © 2011 John Wiley & Sons, Ltd.

Received 8 August 2011; Revised 4 October 2011; Accepted 7 October 2011

KEY WORDS: consensus; software fault localization; program debugging; suspiciousness-based ranking

1. INTRODUCTION

As an area of research, software fault localization, which deals with locating fault(s) in a program once program failure has been observed, has received significant attention over the last few years. This essentially stems from the fact that software fault localization (hereafter referred to simply as 'fault localization' for brevity), has been recognized to be one of the most expensive activities in program debugging [1]. Additionally, to attempt this process manually is time-consuming, tedious, and more gravely, error-prone. In light of such realizations, a host of fault localization techniques have been proposed recently, each of which aims to guide programmers to the locations of faults in one way or another [2–12]. Several such techniques do so via a dynamic analysis of a faulty program, employing execution traces that are collected at runtime. By comparing and contrasting successful and failed execution traces against one another and analyzing the resultant information in various ways, program components (such as functions, blocks, statements, etc.) can be prioritized (i.e., ranked) in order of their likelihood of containing faults. This 'likelihood of containing faults' can be viewed, and is hereafter referred to, as the suspiciousness of the program component.

Once these program components have been ranked in order of their suspiciousness (from most suspicious to least), programmers may then examine them starting from the top of the ranking to

*Correspondence to: W. Eric Wong, Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75080-3021, USA.
†E-mail: ewong@utdallas.edu

assess if a component is faulty or not. A good fault localization technique should place a faulty component high (if not the highest) in its ranking, such that the fault is discovered early in the examination and with minimal effort. An underlying assumption is that of perfect fault detection — a programmer will always correctly classify a faulty component as faulty, and by the same token, a nonfaulty component as nonfaulty. Should this assumption not hold, then the actual number of components that a programmer needs to examine may increase.

Some fault localization techniques generally produce better rankings than others when evaluated against the same faulty program(s), which is indicative of a technique that is generally superior in its construction and/or logic. However, the demonstration of such superiority is still very data-dependent, and the quality of claimed superiority can vary greatly based on which faulty programs are studied. Even if a fault localization technique α performs better (we formally define what we mean by 'performs better' subsequently in Section 3.4 of this paper) than technique β on the majority of the studied data set; it would still mean that β is at least as good as α, if not better, on the remainder of the data set. To the best of our knowledge, no fault localization technique exists that is provably superior to all others, and therefore, shall perform better on all data sets.

This prompts us to wonder why the norm *de rigueur* should be to apply only one fault localization technique, when there are in fact several such techniques that essentially operate on the same set of input data. It instead makes sense to incorporate multiple techniques in some combination, such that advantages of each technique are maintained while the deficiencies are collectively checked. We emphasize that such an endeavor is novel in the context of software fault localization. The problem we are then faced with is how various fault localization techniques may be combined to generally provide a better result. One way to do this would be to analyze how each fault localization technique has been designed, and develop a new technique that satisfies (as best as possible) the requirements and properties of each individual technique. Another strategy would be to take the rankings produced by each technique (each ranking is a listing of the same set of program components, in a potentially different order), and combine them into a consolidated ranking. This paper is pursuant to the second strategy (further discussion as to 'why' is provided in Section 5.2), that is, effectively consolidating — to an agreement — the rankings produced by different fault localization techniques to create one single *consensus-based* ranking[‡] for better fault localization. We summarize the contributions made by this paper as follows:

- A novel and extensible consensus-based fault localization strategy is proposed that makes use of multiple fault localization techniques, as opposed to just one.
- On the basis of case studies performed on the seven programs of the Siemens suite, and the gzip, grep, make, space, and Ant programs, the effectiveness of the proposed strategy is demonstrated.
- Also, via our case studies that made use of the Tarantula, Ochiai, and H3 techniques (see references [2, 5, 9], respectively), an initial set of fault localization techniques suitable for consensus is presented.

The remainder of the paper is organized as follows: Section 2 presents preliminary discussions on fault localization and rank aggregation, which are the two fundamental components of our research. Then Section 3 goes on to describe the design of our experiments, whose results are subsequently presented in Section 4. Section 5 carries out a detailed discussion on issues relevant to the proposed strategy, and the identified threats to validity. Related work is overviewed in Section 6, followed by our conclusions in Section 7.

2. PRELIMINARIES

We first illustrate how fault localization techniques may be used to produce rankings of program components, and then describe rank aggregation in greater depth and outline the procedure based on which the rankings produced by different fault localization techniques are consolidated in this paper.

[‡]Such an endeavor, commonly referred to as 'rank aggregation', while novel in our own context, has been frequented before in a number of different domains as discussed in Section 2.2.

			Coverage					Suspiciousness based on Tarantula	Suspiciousness based on H3
Stmt. #	Desired Program (***P***)	Faulty Program (***P'***)	t_1	t_2	t_3	t_4	t_5		
1	read(a);	read(a);	•	•	•	•	•	**0.5**	**-1.2**
2	read(b);	read(b);	•	•	•	•	•	**0.5**	**-1.2**
3	read(choice);	read(choice);	•	•	•	•	•	**0.5**	**-1.2**
4	if(choice == "sum")	if(choice == "sum")	•	•	•	•	•	**0.5**	**-1.2**
5	print (a + b);	print (a - b);	•	•	•	•		**0.57**	**-1.1**
6	else print (a * b);	else print (a * b);					•	**0**	**-1**
			S	S	S	F	S		

Figure 1. An example to illustrate fault localization (Tarantula is better than H3).

2.1. *Fault localization: an illustrative example*

For the purposes of this paper, we focus on dynamic fault localization techniques that make use of information collected as a result of test case executions against the program under test. More precisely, data on the execution result (success or failure[§]) and the coverage (which statements[¶] are covered/executed by each test case and which are not) are utilized to assign a suspiciousness value to each statement (interpreted as the likelihood of that statement being faulty), which in turn is the basis for ranking the various statements.

Let us consider an implemented program *P'* which differs from the intended target program *P* such that of five test cases in a test set T, one of them results in a failure. Refer to Figure 1 for details. The faulty statement in this example is statement 5 and the failed test case (which is how we know that the implemented program is faulty) is t_4.

The statement coverage information for each test case is also provided: a black dot in the row corresponding to a statement implies that the statement is covered by the test case, and the absence of a black dot means that the statement is not covered by that test case (for example, statement 5 is covered by all tests except t_5). The execution result for each test case is also provided where an 'F' means that the execution resulted in failure and an 'S' means that it was successful. On the basis of the coverage information and the execution results (success or failure), the suspiciousness computed according to the Tarantula and H3 techniques, respectively (see Section 3.1) is listed in the last two columns. Tarantula's statement ranking (based on the suspiciousness) would have statement 5 at the top, followed by statements 1–4, which are tied for the same position in the ranking (because they have the same suspiciousness), and then finally statement 6. This means that a programmer would examine statement 5 before examining any other statements (because it is at the top of the ranking) and would therefore find the fault by examining just one statement. This example also serves to underscore the importance of fault localization. However, the ranking based on H3 would put statement 6 at the very top, and not statement 5. Thus, based on this example it seems that Tarantula is more effective at locating faults than H3.

Now consider a similar example in Figure 2 where the error is in statement 3 (we accidentally multiply 'a' by 'b' instead of dividing). As before, the coverage and execution result information has been provided. We note that in this scenario Tarantula assigns the highest suspiciousness to statement 5 (which is nonfaulty) and not statement 3 (which is faulty). Therefore, statement 5 would be examined before statement 3, were Tarantula used. On the other hand, H3 is clearly able to label statement 3 as more suspicious than statement 5, implying that the fault would be located faster, were H3 used instead of Tarantula. In this case H3 is the better fault localization technique.

[§]A test case is considered successful if the program behavior/output, when executed against the test case, is as expected. Correspondingly, if the observed program behavior/output on a test case deviates from the expected behavior/output, then the test case is a failed test case.

[¶]In this paper we consider 'statements' as the program components of interest with the understanding that without loss of generality, fault localization techniques are equally applicable when considering other program components such as functions, blocks, etc.

Stmt. #	Desired Program (*P*)	Faulty Program (*P'*)	Coverage t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	Suspiciousness based on Tarantula	Suspiciousness based on H3
1	read(a);	read(a);	•	•	•	•	•	•	•	•	•	0.5	0.6
2	read(b);	read(b);	•	•	•	•	•	•	•	•	•	0.5	0.6
3	int c = a / b;	int c = a * b;	•	•	•	•	•	•	•	•	•	0.5	0.6
4	if (b > 10)	if (b > 10)	•	•	•	•	•	•	•	•	•	0.5	0.6
5	c = c + 10;	c = c + 10;		•	•		•	•				0.66	0
6	print (c);	print (c);	•	•	•	•	•	•	•	•	•	0.5	0.6
			F	F	F	S	S	S	S	S	S		

Figure 2. Another example to illustrate fault localization (H3 is better than Tarantula).

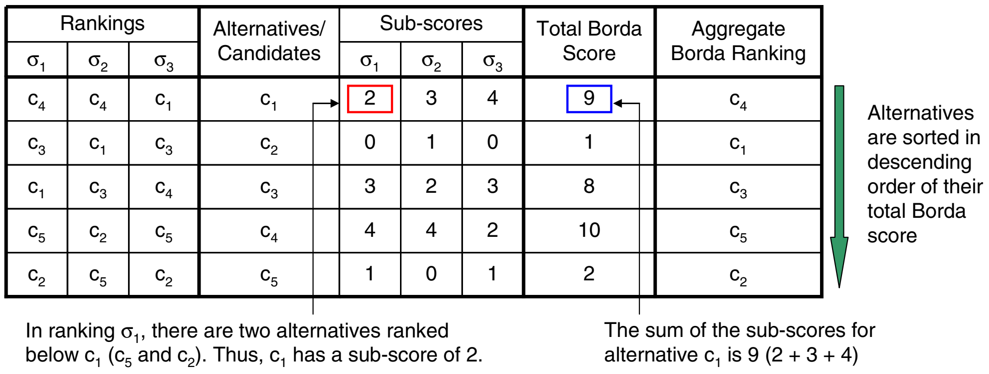
Therefore, if two fault localization techniques (both of good repute) produce statement rankings that are very different from one another (such as Tarantula and H3 in the examples above; Figure 1 and 2), we would be forced to choose which ranking to follow. This may be a bad idea given that no fault localization technique is provably superior to another. However, we note that we would not have to choose one technique over the other, were we somehow able to combine or aggregate their rankings in a sensible way.

2.2. *Rank aggregation*

Informally, the rank aggregation problem is to combine many rank orderings on the same set of candidates or alternatives, to obtain a 'better ordering' [13]. Rank aggregation, in one of its many forms, is seen in different contexts — voters writing orderings of candidates in elections, multiple search engines providing different web-page orderings, etc. The problem is at least as old (and probably much older) as the 18th century, when the Condorcet and Borda voting systems for elections with more than two candidates were proposed [14]. There may be several desirable characteristics of an aggregate ranking depending on the criteria considered. For example, *unanimity* states that if every voter prefers one choice to another, say α to β, then the aggregate ranking should also prefer α to β, that is, rank α above β. Several other criteria exist such as *consistency*, *neutrality*, *monotonicity*, and the *Condorcet criterion* to name a few. Reviewing each of these criteria is beyond the scope of this paper, and thus interested readers are directed to [15] for further details.

One precise criterion based on which an *ideal* aggregate ranking could be determined was provided by Kemeny [14]. Given n different alternatives to choose from, and m potentially different orderings (rankings) of these alternatives $\{\sigma_1 , \sigma_2 \ldots \sigma_m\}$; a *Kemeny optimal* ranking σ is that ordering of the n alternatives, which minimizes $\sum_{i=1}^{m} D(\sigma, \sigma_i)$. Here, $D(\sigma, \sigma_j)$ denotes the number of pairs of alternatives that are ranked differently by σ and σ_j. This is in fact the well-known *Kendall Tau* distance metric. Stating this condition for optimality differently, σ maximizes the number of pair-wise agreements with the votes [16]. Unfortunately, finding a Kemeny optimal ranking is a nondeterministic polynomial-time hard problem and remains nondeterministic polynomial-time hard even when there are as few as four rankings to be aggregated [17].

With this in mind, we choose Borda's method [18] to determine aggregate rankings because this method can be implemented in linear time and has been shown to be a factor-5 approximation to the Kemeny optimal ranking [19] (our choice of the Borda technique is further discussed in Section 5.4). Borda's method works in the following manner: given m complete orderings $\{\sigma_1 , \sigma_2 \ldots \sigma_m\}$ of n alternatives; for each alternative c in each of the orderings σ_i, a subscore $B_i(c)$ is assigned, which is defined as the number of alternatives ranked below c in σ_i. The total Borda score for alternative c is then computed as $\sum_{j=1}^{m} B_j(c)$, that is, the sum of the subscores across each of the orderings. The aggregate ranking is formed by sorting the alternatives in decreasing order of their total Borda scores such that an alternative higher in the ranking is preferred to one lower in the ranking. Following these steps precisely, an aggregate Borda ranking can be generated. An illustration of this process is provided via example in Figure 3.



Rankings			Alternatives/ Candidates	Sub-scores			Total Borda Score	Aggregate Borda Ranking
σ_1	σ_2	σ_3		σ_1	σ_2	σ_3		
c_4	c_4	c_1	c_1	$\boxed{2}$	3	4	$\boxed{9}$	c_4
c_3	c_1	c_3	c_2	0	1	0	1	c_1
c_1	c_3	c_4	c_3	3	2	3	8	c_3
c_5	c_2	c_5	c_4	4	4	2	10	c_5
c_2	c_5	c_2	c_5	1	0	1	2	c_2

Alternatives are sorted in descending order of their total Borda score
In ranking σ_1, there are two alternatives ranked below c_1 (c_5 and c_2). Thus, c_1 has a sub-score of 2.
The sum of the sub-scores for alternative c_1 is 9 (2 + 3 + 4)

Figure 3. An example to illustrate Borda's method of rank aggregation.

With respect to Figure 3, the left-most columns present three different rankings (σ_1, σ_2, and σ_3) of five different alternatives ($c_1, c_2, c_3, c_4,$ and c_5), which are to be consolidated to a single ranking via the Borda method. As per the steps described above, first the subscores for each alternative are calculated (shown in the columns in the middle). For example, we observe that based on ranking σ_1, alternative c_1 has a subscore of 2 because there are two alternatives (c_5 and c_2) ranked below c_1. On the other hand, based on ranking σ_3, alternative c_1 has a subscore of 4 because there are four alternatives (c_3, c_4, c_5, and c_2) ranked below c_1. Next, the total Borda score for each alternative is calculated by summing across the subscores for that alternative. For example, from the figure we see that alternative c_1 has a Borda score of 9 (2+3+4), whereas c_5 has a Borda score of 10 (4+4+2). Finally, the aggregate Borda ranking is formed by sorting the alternatives in decreasing order of their total Borda scores (shown at the very right of the figure). This example serves a dual purpose in that not just does it illustrate the Borda method, but it also shows how the aggregate Borda ranking may be different from the individual rankings that constitute it (as per our example, none of the component rankings is the same as the Borda ranking).

In such a way, different rankings produced by different fault localization techniques may also be aggregated to a single consensus-based ranking. Once the consensus-based ranking has been produced, the process of locating bugs would continue in exactly the same manner as if a singleton fault localization technique had been used, that is, developers would examine the ranking statement by statement, starting from the very top of the ranking. However, the usefulness of such an approach requires empirical evaluation to validate, which is why we conducted a set of comprehensive experiments discussed in the following sections.

3. EXPERIMENTAL DESIGN

In this section we overview the fault localization techniques and subject programs considered for our experiments and provide details on how data was collected. Also discussed is the metric used to evaluate fault localization techniques and any special design considerations relevant to our experiments.

3.1. *Fault localization techniques*

Three fault localization techniques have been chosen for our experiments, namely, Tarantula [5] (mentioned in Section 2.1); a technique based on the Ochiai coefficient [2], hereon referred to as Ochiai; and a technique based on a heuristic that calibrates the contributions made by successful and failed executions [9], referred to by the authors as H3. Each of these techniques was chosen with a purpose in mind — Tarantula, because it is simple and popular, and the others because they are quite different in their internal design, yet both report results that are generally superior to techniques such as Tarantula. Additionally, each of these techniques operates on exactly the same input data (coverage information with respect to a set of program components for each test case and the

Table I. Notations relevant to the fault localization techniques.

N_F	Total number of failed test cases
N_S	Total number of successful test cases
$N_C(\omega)$	Number of test cases covering ω
$N_{CF}(\omega)$	Number of failed test cases covering ω
$N_{CS}(\omega)$	Number of successful test cases covering ω

test execution result — success or failure), and produces as output rankings of the same set of program components. We remind the reader that for the purposes of this paper, program components are considered to be executable statements,[‖] with the understanding that without loss of generality, other components such as functions, blocks, etc., could have just as easily been considered. Hereafter, unless otherwise specified, we refer to executable statements simply as statements. We now present a succinct overview of each of the selected techniques (further discussion on their choice is presented in Section 5.4). To make the descriptions of each technique simpler and to be consistent, we make use of the notations provided in Table I. For any statement ω we have:

Where the total number of test cases is given by $(N_F + N_S)$ because the outcome of a test case is always binary, that is, the execution must either fail or succeed. Also note that the values of N_F and N_S are the same with respect to all statements of the same program.

3.1.1. Tarantula. Tarantula follows the intuition that statements that are executed (covered) primarily by failed test cases are highly probable (more suspicious) to be faulty [5]. Additionally, statements that are executed primarily by passed (successful) test cases are less likely to be faulty. Tarantula thus assigns a suspiciousness value to each statement using the formula:

$$susp(\omega) = \frac{\frac{N_{\mathrm{CF}}(\omega)}{N_{\mathrm{F}}}}{\frac{N_{\mathrm{CS}}(\omega)}{N_{\mathrm{S}}} + \frac{N_{\mathrm{CF}}(\omega)}{N_{\mathrm{F}}}} \quad (1)$$

where $susp(\omega)$ gives the suspiciousness of statement ω. On the basis of a case study performed on the programs of the Siemens suite, it is shown that Tarantula is more effective than other fault localization techniques such as set union, set intersection, nearest neighbor, and cause-transitions. [5]

3.1.2. Ochiai. Following the same convention, the Ochiai coefficient has been utilized to assign statement suspiciousness as follows:

$$susp(\omega) = \frac{N_{\mathrm{CF}}(\omega)}{\sqrt{N_{\mathrm{F}} \times (N_{\mathrm{CF}}(\omega) + N_{\mathrm{CS}}(\omega))}} = \frac{N_{\mathrm{CF}}(\omega)}{\sqrt{N_{\mathrm{F}} \times N_{\mathrm{C}}(\omega)}} \quad (2)$$

It has been shown empirically that Ochiai consistently outperforms other coefficients such as Jaccard [20] and also Tarantula when used for fault localization [2].

3.1.3. H3. H3 (the most efficient of several proposed in [9]) assigns statement suspiciousness as:

$$susp(\omega) = N_{\mathrm{CF}}(\omega) - \Omega(N_{\mathrm{CS}}(\omega)) \quad (3)$$

$$\text{Where } \Omega(k) = \begin{cases} k, & \text{for } k = 0, 1, 2 \\ 2 + (k-2) \times 0.1, & \text{for } 3 \le k \le 10 \\ 2.8 + (k-10) \times \alpha, & \text{for } k \ge 11 \end{cases}$$

The computation involves a variable α, which we set to 0.001 for our experiments, because this is the value that yields the best results based on the experiments performed in [9]. These

[‖]Multiline 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 [5]. What we are then left with are executable statements.

experiments also reveal that H3 shows significant improvement over Tarantula in terms of its fault localization capabilities.

3.2. Subject programs

The seven programs in the Siemens suite have been used extensively in testing and fault localization related studies such as those in [2, 4, 6, 7, 9, 10, 21]. The correct versions, 132 faulty versions, and all the test cases were downloaded from [22]. Three faulty versions are excluded in our study: version 9 of 'schedule 2' 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. Table II presents further details regarding the programs of the Siemens suite.

The source code of version 2.2 of the grep program was downloaded from [23]. Also downloaded were versions 1.1.2 of the gzip program and 3.76.1 of the make program, and the space program. Because these are all categorically C programs (i.e., programs written in the C programming language), and we wished to evaluate object-oriented programs as well, we also downloaded version 1.6 beta of the Ant program from [23]. Each of these programs (correct versions) was downloaded with a set of accompanying test cases and faulty versions. For four programs (grep, gzip, make, and Ant), we also created faulty versions using mutation-based fault injection, in addition to the ones that were downloaded. This was made to enlarge our data sets and because mutation-based faults have been shown to simulate realistic faults well, and provide reliable and trustworthy results [6, 24–26]. For the grep program, additional faults from [23] were also used. Finally, for all of our subject programs, any faulty versions that did not lead to at least one test case failure in our execution environment were excluded. A summary of the information regarding grep, gzip, make, space, and Ant is presented in Table III.

We observe that the gzip, space, grep, and make programs are much larger (in terms of the LOC count) than the programs of the Siemens suite. In fact the make program is considerably larger than the grep program, which in turn is larger than the gzip program. The Ant program is significantly larger than all of these programs. This allows for a much more comprehensive evaluation across not just different programs, but also programs of varying sizes: small to very large.

3.3. Data collection

All program executions were on a PC with a 2.13 GHz Intel Core 2 Duo CPU and 8 GB physical memory. The operating system was SunOS 5.10 (Solaris 10, Oracle Corporation, California, USA) and the compiler used was GCC 3.4.3 (Free Software Foundation, Boston, MA, USA). Each faulty

Table II. Summary of Siemens suite.

Program	Number of faulty versions used	Number of test cases	Number of LOC
print_tokens	5	4130	565
print_tokens2	10	4115	510
schedule	9	2650	412
schedule2	9	2710	307
replace	32	5542	563
tcas	41	1608	173
tot_info	23	1052	406

LOC: lines of code.

Table III. Summary of grep, gzip, make, space and Ant programs.

Program	Number of faulty versions used	Number of test cases	Number of LOC
grep	19	470	12,653
gzip	28	211	6573
make	31	793	20,014
space	35	13,585	9126
Ant	23	871	75,333

version was executed against all its corresponding available test cases. The success or failure of an execution was determined by comparing the outputs of the faulty version and the correct version of a program. A failure was recorded whenever there was deviation of observed output/behavior from expected output/behavior. The tool χSuds (Telcordia Technologies, Piscataway, NJ) [27] was used to collect execution trace (statement coverage in our case) information for the C programs, while Clover (Atlassian Corp, Sydney, Australia) [28] was used in the case of the Java-based Ant program.

3.4. *Evaluation metric*

Renieris and Reiss [7] suggested the assignment of a score to each faulty version of a subject program, 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 in [4]. Tarantula operates directly on the program's source code and to make their computations comparable to those based on the PDG, the authors consider only executable statements in the computation of their score [5]. Thus, for consistency, only executable statements are considered for our experiments as well.

However, one difference between the above measures and the one used in this paper is that we think it is more straightforward and suitable to report the effectiveness in terms of the percentage of code that must be examined to find the fault, as opposed to that which need not. This modified score is referred to as the $\mathcal{EXAM}$ score [9–11] and is defined as the percentage of executable statements that must be examined (starting from the top of the ranking) until a faulty statement is reached. A similar modification is made by Liu *et al.* [6] with the exception that their score is again based on the PDG. Note that this $\mathcal{EXAM}$ score and the one used in [5] are equivalent and can be derived one from the other. In terms of comparing techniques: for any given faulty program, if the $\mathcal{EXAM}$ score assigned to it by a fault localization technique $\mathscr{L}_\alpha$ is less than that assigned by a different technique $\mathscr{L}_\beta$, then we conclude that $\mathscr{L}_\alpha$ outperforms or is better than (i.e., is more effective than) $\mathscr{L}_\beta$ (as fewer statements need to be examined to find a fault).

Because the $\mathcal{EXAM}$ score (and other such scores) can only be used to compare two or more techniques on the same program, and we in fact have multiple faulty versions of each program under study; we also compare from a more cumulative perspective. Supposing we have k faulty versions as part of a dataset, and $\mathscr{L}_\alpha(i)$ and $\mathscr{L}_\beta(i)$ give the total number of statements that must be examined to locate a fault on the i^{th} faulty version, by techniques $\mathscr{L}_\alpha$ and $\mathscr{L}_\beta$ respectively, we can say $\mathscr{L}_\alpha$ is more effective than $\mathscr{L}_\beta$, if $\sum_i^n \mathscr{L}_\alpha(i) < \sum_i^n \mathscr{L}_\beta(i)$.

3.5. *Ties within rankings*

Prior to presenting the results, it is necessary to point out another aspect of the suspiciousness computation — it is possible that the same suspiciousness value may be assigned to different statements. Stated differently, the suspiciousness that is assigned to a statement may not necessarily be unique, and therefore two (or more) statements may be tied for the same position in the ranking. For the fault localization techniques used in this paper one way (though not the only way) for this to happen is if two (or more) statements have the same values for each of the attributes listed in Table I.

With regards to consensus, often in voting scenarios when voters are unable to select between two or more candidates (alternatives), these candidates are ranked based on some *key* or natural ordering. For example, an alphabetical ordering to break ties. Along the same lines, between two statements tied for the same ranking, we ranked the statement with a lower statement number above that with a higher statement number. Note that these statement numbers only correspond to the order in which the statements appear in the program, and are not necessarily the order in which the statements are executed. For example, when a source file is opened in a text editor, the line numbers assigned to statements (by the text editor) can serve as our key. Even in the case of multiple source files, as long as two statements (whether in the same file or not) are not assigned the same statement number, ties can be broken.

To ensure fairness, we use this same key (i.e., statement number) to break ties in the rankings produced by Tarantula, Ochiai, and H3, and in the final ranking produced via Consensus (computed as per Borda's method as described in Section 2.2). However, the use of such a key is purely to break any ties in the fault localization rankings, and in fact is not expected to have any significant bearing on the actual quality of fault localization (by any of the techniques). We return to this issue later in the paper when we discuss the identified threats to validity in Section 5.4.

4. EXPERIMENTAL RESULTS

Figures 4–9 show the fault localization effectiveness comparison across the Siemens suite, the grep, gzip, make, space, and Ant programs, respectively. Three curves in the figures are annotated as 'Ochiai,' 'H3', and 'Tarantula', each signifying the effectiveness of its corresponding fault localization technique. The fourth curve, representing the effectiveness based on the rank aggregation procedure detailed in Section 2.2, is annotated as 'Consensus'. This same convention is followed for all of the figures. In addition to being patterned differently and having different point markers, these curves are also displayed in different colors and hence are best viewed in color.

In each figure, the x-axis represents the percentage of code that is examined (the $\mathcal{EXAM}$ score) and the y-axis represents the percentage of faults located for the corresponding $\mathcal{EXAM}$ score. For example, from Figure 4 (programs of the Siemens suite) we observe that 62.79% (81/129), 51.16% (66/129), 58.91% (76/129), and 66.67% (86/129) of the faults can be located, by examining less than 10% of the code when using the consensus-based, Tarantula, Ochiai, and H3 techniques, respectively. To enhance readability, zoom-ins have been provided at the bottom right of each of the figures, for when up to a small percentage of the code is examined.

The first thing we observe is that there is no singleton fault localization technique that seems to dominate across the entire data set, that is, the six sets of programs studied. For example, H3 seems to be the best technique with respect to the Siemens suite (as per Figure 4) and Ant (as per Figure 9), yet this is not necessarily the case for the other programs. This observation supports the claim that using one fault localization technique alone may not always be the best strategy, as the quality of fault localization can vary drastically from data set to data set (i.e., from program to program). We also note that Tarantula seems to be the worst of the singleton techniques, and this is especially evident based on the Siemens suite and gzip programs (Figure 4 and 6, respectively). This observation shall form the basis of the discussion in Section 5.1.

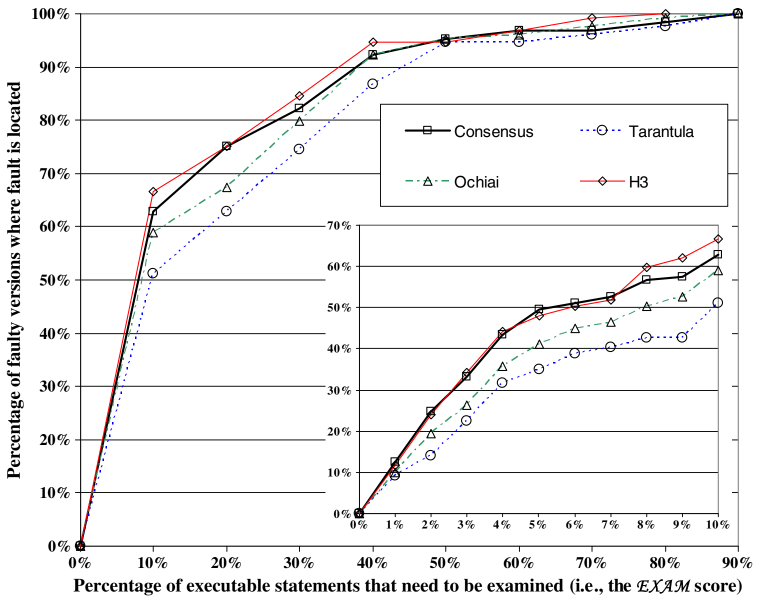


Figure 4. Effectiveness comparison on the Siemens suite.

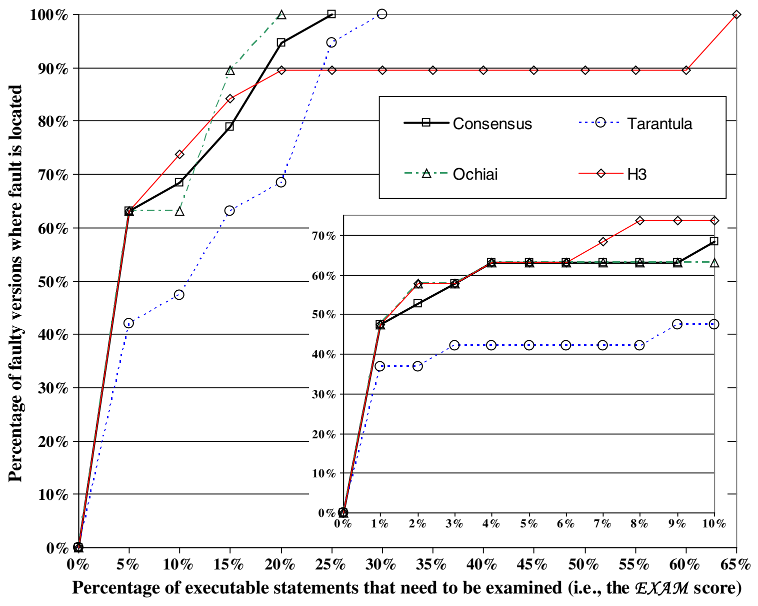


Figure 5. Effectiveness comparison on grep.

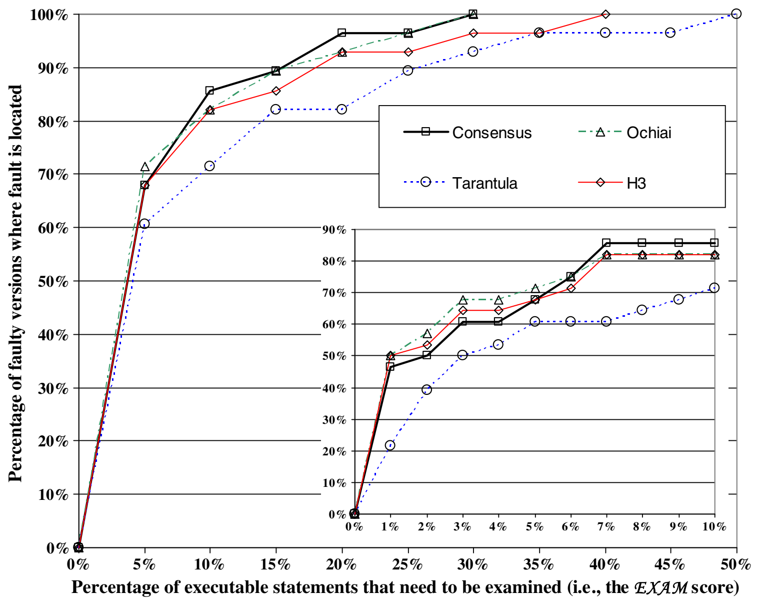


Figure 6. Effectiveness comparison on gzip.

We digress temporarily to now fully justify what we mean by 'better' and 'worse' performance by the techniques. Note that the curves intersect each other quite often signifying that the effectiveness of one technique may be better or worse than the other depending on what percentage of the code has been examined. For example, H3 on the grep program leads to the localization of 89.47% (17/19) of the faults by examining less than 20% of the code. However, as far as the remaining 10.53% (2/19) of the faults are concerned, H3 can only locate these faults by examining about 65% of the code. In contrast, Tarantula, behind H3 initially, now surpasses H3 as it locates these 2 faults before H3. To avoid setting arbitrary thresholds as to 'how much code *should* be examined' and to keep comparisons fair, in such a scenario we consider the technique that locates all the faults by examining the least amount of code to be superior.

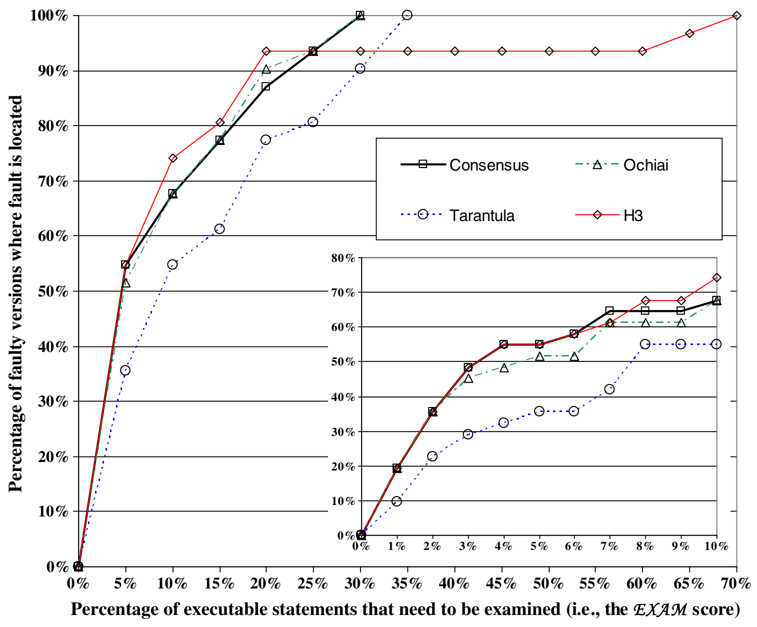


Figure 7. Effectiveness comparison on make.

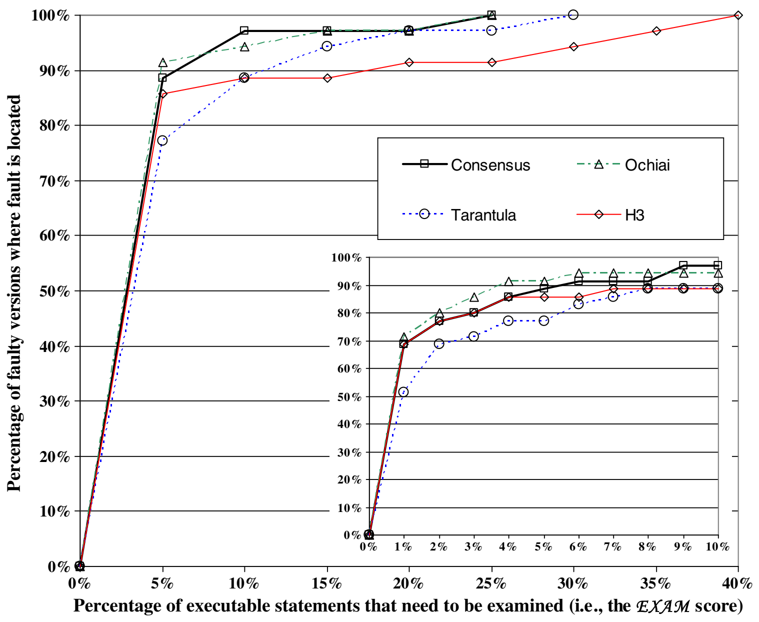


Figure 8. Effectiveness comparison on space.

Back to our discussion on Consensus, the second arguably more pertinent observation is that Consensus performs fairly well across all the data sets studied. We use the word ‘fairly’ because even if Consensus does not deliver the best fault localization effectiveness in every case, its performance, if not the best, is always close to the best observed. This relatively consistent yet high quality performance across all the data sets studied is indicative of its usefulness as a general strategy to improve the effectiveness of fault localization.

Given that the curves in Figures 4–9 may not be perfectly decipherable to all, and that the $\mathcal{EXAM}$ score only provides one frame of reference for comparison, Table IV presents data in terms of the total number of statements that need to be examined to locate all the faults used in our studies (as discussed at the end of Section 3.4) per fault localization technique. We emphasize that each

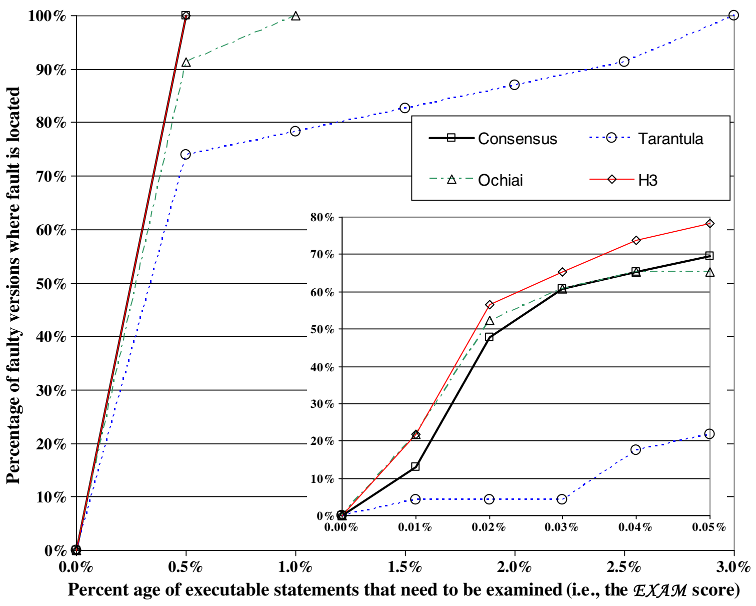


Figure 9. Effectiveness comparison on Ant.

Table IV. Total number of statements examined to locate all faults for the subject programs.

	Siemens suite	grep	gzip	make	space	Ant
Tarantula	2706	6553	4060	19,951	4490	6505
Ochiai	2063	3566	2155	12,770	2145	1248
H3	1657	6358	2601	15,489	5115	444
Consensus	1778	3937	2120	12,219	2636	557

entry in the table corresponds to the aggregate number of statements that each technique requires the examination of to locate faults in *all* of the faulty versions per subject program (as opposed to per faulty version). The average number of statements to be examined per faulty version can easily be computed by dividing each entry by the number of faulty versions (corresponding to that program). For example (based on Table IV), we see that Consensus requires the examination of 1778 statements on the Siemens suite, which corresponds to only about 13.78 statements (1778/129) per faulty version.

As in the case of the $\mathcal{EXAM}$ score-based comparison, we note that there is no singleton technique that performs the best in all cases. This provides clear evidence in support of a strategy that takes into account multiple fault localization techniques, such as Consensus. In fact in the case of the gzip and make programs, we note that Consensus can locate all the faults by the examination of no more than 2120 and 12,219 statements, respectively, numbers that are lower than those corresponding to any of the singleton techniques. As we noted with respect to the $\mathcal{EXAM}$ score, when Consensus cannot provide the best performance, it is always close to the best. For example on the space program, while Ochiai is the best technique, Consensus is much closer to Ochiai than Tarantula or H3. On the other hand on the Ant program, when H3 happens to be the best technique, Consensus is still much closer to H3 than Tarantula or Ochiai.

Thus, the data in Table IV, along with Figures 4–9, indicates that Consensus indeed provides high quality fault localization effectiveness even across different data sets.

5. DISCUSSION

In this section, we present discussions on issues and concerns that are relevant to the consensus-based strategy presented in this paper and the results that have been so derived. Also discussed in this section are the identified threats to validity.

5.1. Research questions regarding inclusion/exclusion of techniques from consensus

An important concern regarding consensus applied to our context is the quality or goodness of the fault localization techniques that are used for the purposes of consensus. Intuitively, we would like to include good techniques in the consensus and exclude the bad (relatively worse) ones, because the net effect of including lesser techniques is to generally bring down the consensus result. However, throughout the paper we have emphasized that it is very rare (if not entirely impossible) for any one technique to provide excellent fault localization results on every data set, and indeed even if one technique seems to be better than another most of the time, this may not be true all of the time. Thus, it is difficult to perfectly categorize one technique as good (or bad) with respect to another. In such a situation a popular strategy would be to include a large and diverse assortment of techniques for consensus, such that the benefit of each technique is reflected in some way in the consensus. We are therefore confronted with two distinct, yet indirectly related, research questions (RQs) regarding the consensus-based strategy that has been proposed, namely:

RQ1: Is including more fault localization techniques in the consensus always recommended?
RQ2: Is excluding a (reportedly) less effective fault localization technique from consensus always recommended?

Both questions can be addressed somewhat simultaneously by adding another fault localization technique into our consensus, which has a poor performance relative to the other techniques. However, a simpler way to achieve the same would be to remove Tarantula from the consensus, as this would reduce the number of techniques used by one; and additionally, the other techniques, H3 [9] and Ochiai [2], have reported results that are generally superior to Tarantula (though not necessarily in all cases).

Figures 10–15 show the effectiveness comparison between consensus using all three fault localization techniques discussed in Section 3.1 (annotated as Consensus), and consensus using just the H3 and Ochiai techniques (annotated as Consensus-New). Because the curves for H3 and Ochiai themselves undergo no change (from Figures 4–9) and are immaterial to this comparison, they have been left out. The curves for Consensus too undergo no change but they are still shown in the figures for ease of reference. As before, zoom-ins have been provided for each of the figures, except in the case of the Ant program (referring to Figure 15) where the scale of the graph is already very low.

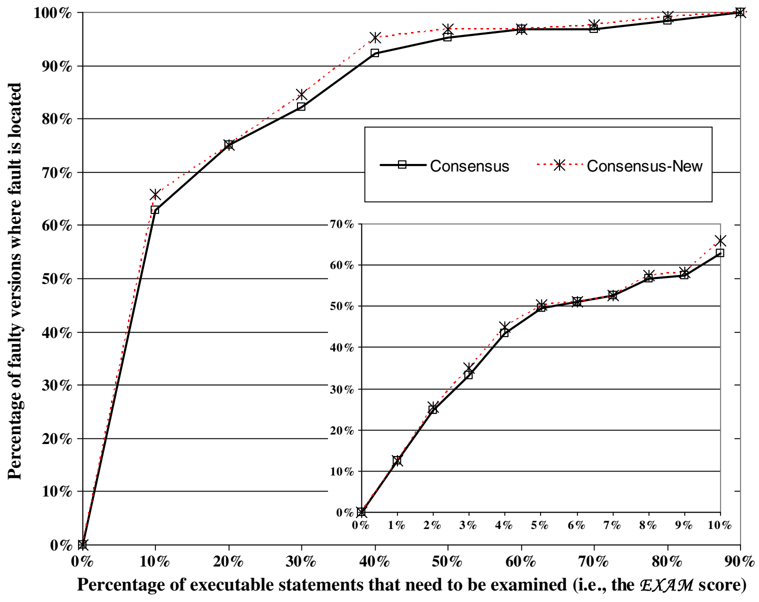


Figure 10. Effectiveness comparison on the Siemens suite.

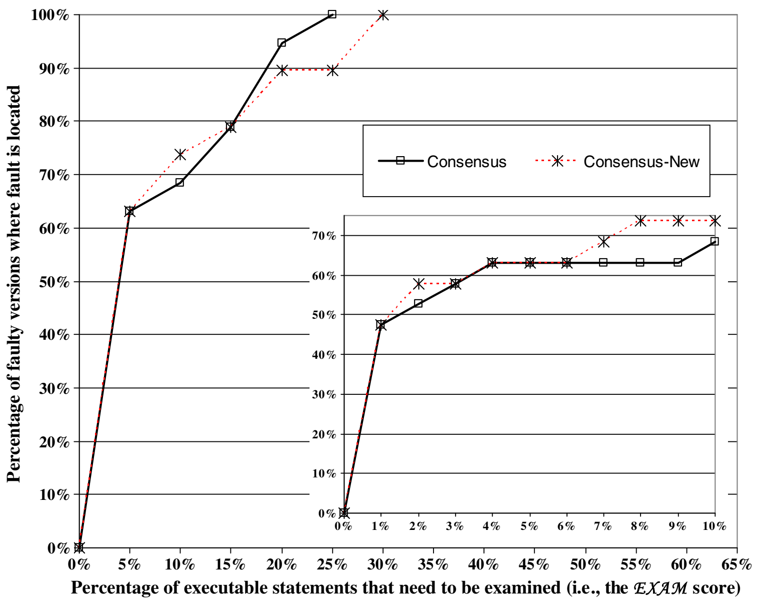


Figure 11. Effectiveness comparison on grep.

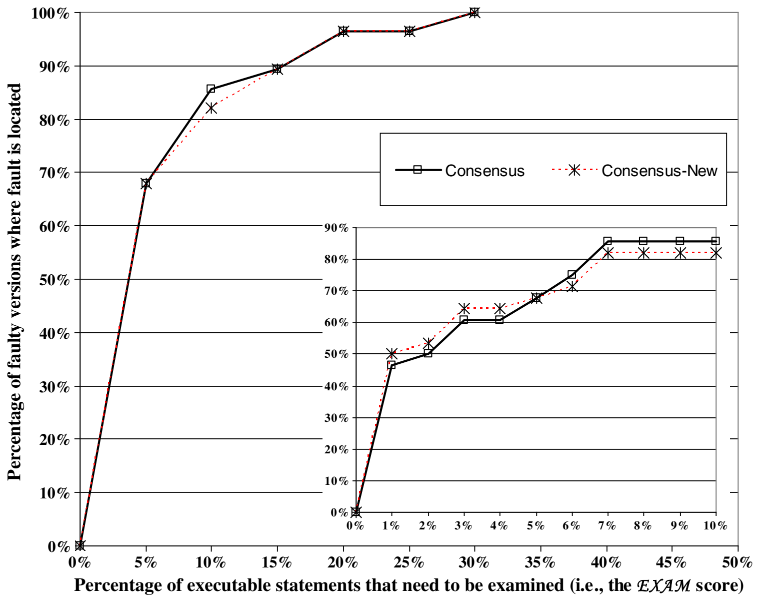


Figure 12. Effectiveness comparison on gzip.

While Consensus-New seems to offer improvement over Consensus on the programs of the Siemens suite (Figure 10) and the Ant program (Figure 15), this is not the case for the other programs (Figures 11– 14). In fact on the space program (Figure 14) Consensus-New clearly performs worse than Consensus. Similarly, on the grep program (Figure 11) Consensus-New seems to do worse than Consensus because Consensus can locate all of the faults by examining less than 25% of the code, whereas Consensus-New can only locate 89.47% (17/19) faults by examining up to 25% of the code. Notably though, if we examine only 10% of the code, then Consensus-New is better than Consensus on the grep program as per the zoom-in in Figure 11. Additionally, Table V presents a comparison between Consensus and Consensus-New based on the total number of statements that need to be examined to locate all the faults used in our study. Similar to the figures, Table V reveals

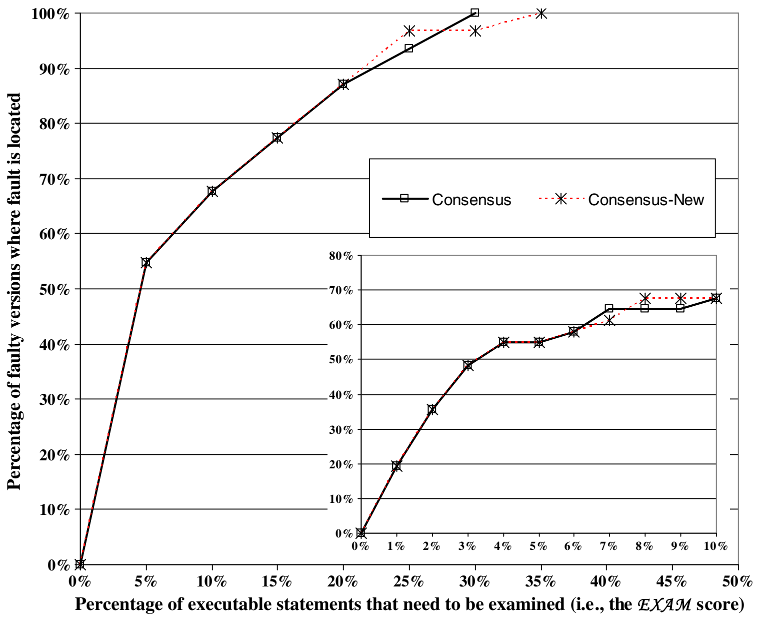


Figure 13. Effectiveness comparison on make.

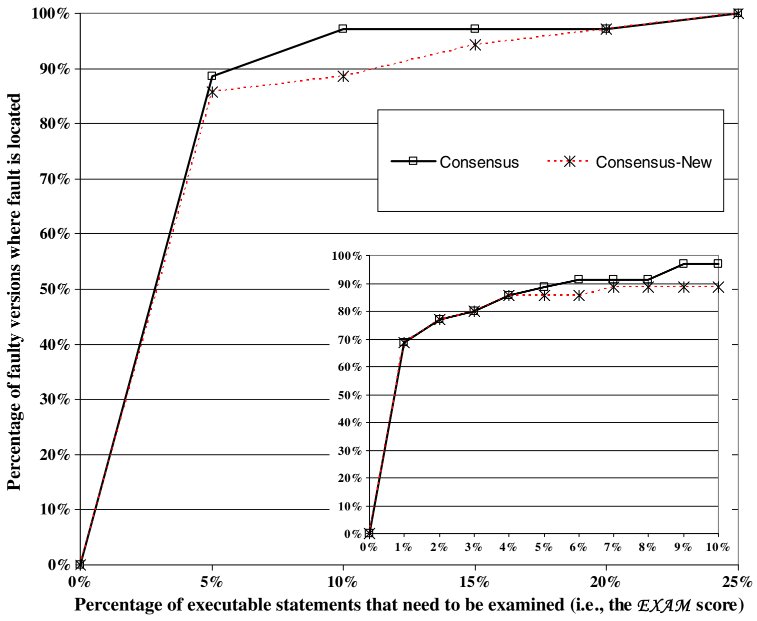


Figure 14. Effectiveness comparison on space.

improvement by using Consensus-New as opposed to Consensus, but only on the programs of the Siemens suite and Ant. On the grep, gzip, make, and space programs, more statements need to be examined to locate all the faults by using Consensus-New (than Consensus).

On the basis of the data in Figures 10–15 and Table V, we observe that removing Tarantula from the consensus does not really improve upon the results (the results are only improved in the case of the Siemens suite of programs and the Ant program), and in fact seems to bring down the overall quality of fault localization (this is especially evident from Table V). The reason for this is exactly what motivated the consensus-based strategy for fault localization in the first place. Even though the other fault localization techniques have been reported to be more effective than Tarantula, this

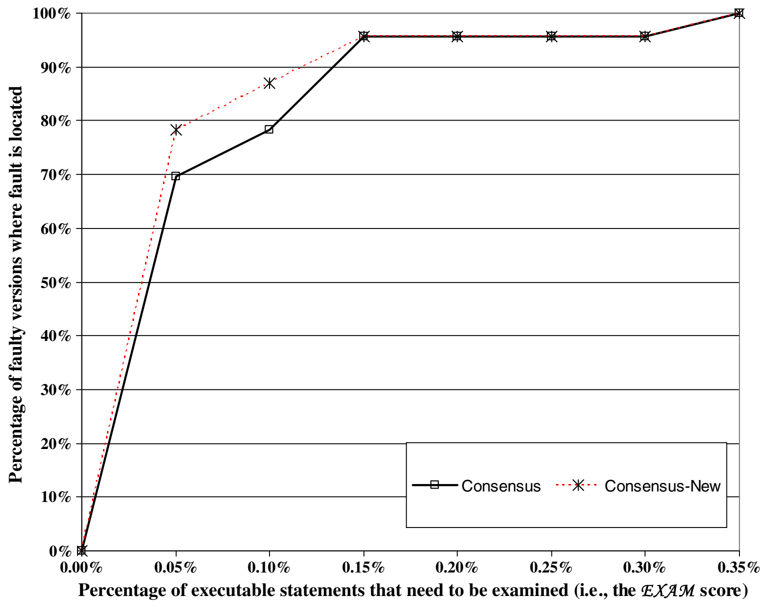


Figure 15. Effectiveness comparison on Ant.

Table V. Total number of statements examined to locate all faults for the subject programs.

	Siemens suite	grep	gzip	make	space	Ant
Consensus	1778	**3937**	**2120**	12,219	**2636**	557
Consensus-New	**1620**	4283	2128	12,541	3192	**453**

is not always the case. Consider the grep program (Figure 5), where the performance (in terms of the percentage of code examined for locating all the faults used in our study) of Tarantula is not as poor, when compared with the H3 technique. Retaining H3 and excluding Tarantula (from consensus) in this case, leads to a worse performance than had we just included both of them along with Ochiai. The same is true of the gzip and make programs, though not to the same extent as in the case of the grep program. Note that we can still be confident in the claim that our consensus-based strategy provides close to the best fault localization results, if not the best (as can be observed via a simple comparison of the data in Tables IV and V — by comparing the data corresponding to Consensus-New with that of Ochiai and H3).

We are now in a position to address the two research questions from before. The answer to RQ1 seems to be no. As seen on the Siemens suite and Ant (see Figures 10 and 15 and Table V), having a fewer number of fault localization techniques involved in the consensus can lead to better results. On the other hand, for the grep, gzip, make, and space programs, having a fewer number of fault localization techniques leads to a worse result. Hence, in no way we can conclude that selecting a smaller (or a larger) set of fault localization techniques always gives a better result.

Research question 2 is much trickier to answer. The choice of whether to include a less effective fault localization technique really depends on our definition of 'less effective'. Unless a fault localization technique is provably inferior (or inferior beyond contention) to others in every case, it may still be valuable to the consensus. Excluding Tarantula leads to worse results in the case of some programs as seen in Table V. Thus, prior to rejection, we should be highly confident in the ability of the other techniques to outperform what we believe is the 'less effective' technique. Even though fault localization techniques such as H3 and Ochiai reportedly are more effective than Tarantula, our data suggests that removing Tarantula from Consensus does not necessarily imply that the fault localization results will always improve. Therefore, techniques such as Tarantula may still be very useful from the point of view of fault localization (especially if a consensus-based approach is followed).

Finally, to close this section, we reemphasize that the goal of consensus is to have multiple fault localization techniques work together to produce a better result (in our study, either the best or close to the best), such that the strengths of each technique are maintained and the deficiencies collectively checked. Essentially, even if a fault localization technique we use does not turn out to be as effective as we had thought, the other techniques in the consensus should prevent it from bringing down the overall effectiveness. Thus, we have a flavor of *quis custodiet ipsos custodes* (Latin for 'who watches the watchmen', which has many similar interpretations, one of the more apt ones being 'who will guard us from the guardians').

5.2. Design-based versus ranking-based consensus

As mentioned before, a consensus between the fault localization techniques may be achievable either by consolidating rankings (as has been done in this paper) or by the consolidation of the techniques themselves, perhaps by finding commonalities in their design. The latter approach has neither been pursued by us, nor is it recommended to others (whether in research or in practice), as it is fraught with complications and may not be a practical option.

Consider that a fault localization technique is essentially a function $\mathtt{L}$ that takes (in our case) as input (x), statement coverage and test execution result information, and transforms that information into a ranking of statements (r). Thus, we have $\mathtt{L}(x) \to r$. Given two techniques, $\mathtt{L}_\alpha$ and $\mathtt{L}_\beta$, we may have two potentially different rankings as $\mathtt{L}_\alpha(x) \to r_\alpha$ and $\mathtt{L}_\beta(x) \to r_\beta$. Merging these two functions to produce a new function $\mathtt{L}$ is impossible without detailed knowledge about each technique (and this merger may never even be possible as some techniques may be contradictory in design). In the event that we wish to add another technique to the consensus, the consolidated technique will have to be remodified in its construction. This can be extremely time-consuming and tedious. The same applies if we want to remove a technique from the consensus.

In contrast, working with the rankings does not suffer from any of the above problems. To add another fault localization technique to the consensus is as simple as including its produced ranking in the aggregation. By the same token, the removal of a technique from the consensus requires no more than just discounting its ranking. No detailed analysis is required and we are able to treat each technique as a black-box. Indeed, all the consensus-based strategy proposed here-in requires is the set of rankings to consolidate, and really there is no need to even know which techniques have been used. The ability of such consensus to easily accommodate new techniques makes it highly extensible, yet its ability to also remove techniques with ease makes it tractable, both of which are desirable characteristics of such strategies. Furthermore, there are several ways to combine rankings, of which the Borda method followed in this paper (see Section 2.2) is but only one, which means that the proposed consensus-based strategy offers a great deal of flexibility and customizability in how the (ranking-based) consensus is actually achieved. At this point we reemphasize that Borda's method was chosen because it provides a factor-5 approximation to the Kemeny optimal ranking and can be implemented in linear time. The additional time required to run our consensus-based approach as opposed to using a singleton technique is trivial (and in our execution environment, the extra time required is on the order of a few milliseconds per faulty program).

5.3. Single versus multifault programs

The faulty versions of the programs studied have all categorically been single fault versions, that is, each faulty version has exactly one fault in it. However, it is important to point out that the proposed strategy does not represent a technique by itself, but rather calls for the use of multiple techniques in tandem, irrespective of the number of faults in a program. Thus, its use is in no way restricted to single-fault programs and it can very easily be applied to programs with multiple faults as well. To evaluate how well the Consensus-based strategy fares in a multi-fault setting, we conducted an additional set of experiments.

For programs with multiple faults in them, the authors in [29] defined an evaluation metric that they termed as 'Expense' corresponding to the percentage of code that must be examined to locate the first fault as they argue that this is the fault that programmers will begin to fix. We note that the 'Expense' score, though defined in a multifault setting, is very similar to the $\mathcal{EXAM}$ score used in this paper. Thus, fault localization techniques can also be applied to and evaluated on programs with multiple faults in such a manner, and in accordance we conduct a comparison between our Consensus-based strategy and other fault localization techniques using this approach. As per [29] all the techniques (and Consensus) are evaluated on the basis of the 'expense' required to find only the first fault in the ranking. Note that because the rankings based on the various fault localization (including Consensus) techniques may vary considerably with respect to one another, it is not necessary that the first fault located by one technique be the same as the first fault located by another technique.

Multifault versions of the subject programs are created via a combination of several single-fault versions that are available to us. For example, a two-fault version of a program can be created by seeding each of the faults of two single-fault versions of this program into the correct version. The programs of the Siemens suite have been used for the purposes of this comparison as there are many single-fault versions available (see Table II in Section 3.2), which can be combined in a variety of ways to produce many different multifault versions. A total of 75 such programs are created based on combinations of the single-fault programs of the Siemens suite, ranging from faulty versions with two faults in them to those with five faults in them.

Figure 16 presents the comparison between Consensus and the other fault localization techniques on the multifault programs of the Siemens suite. The figure clearly reveals that Consensus performs very well, much better in fact when compared with the singleton fault localization techniques. We also present the total number of statements that need to be examined to find the first fault across all 75 faulty versions studied, in Table VI. From the table it is clear that a Consensus-based strategy such as the one evaluated herein is superior to the use of any fault localization technique by itself.

In summary, our experiments reveal that the Consensus-based strategy is not just equally applicable to single-fault and multifault scenarios alike, but rather has the potential to be highly effective in either scenario as well.

5.4. Threats to validity

A threat to the construct validity is the use of the $\mathcal{EXAM}$ score as a mode of comparison between different fault localization techniques. Our evaluation metrics in Section 3.4 may not provide a

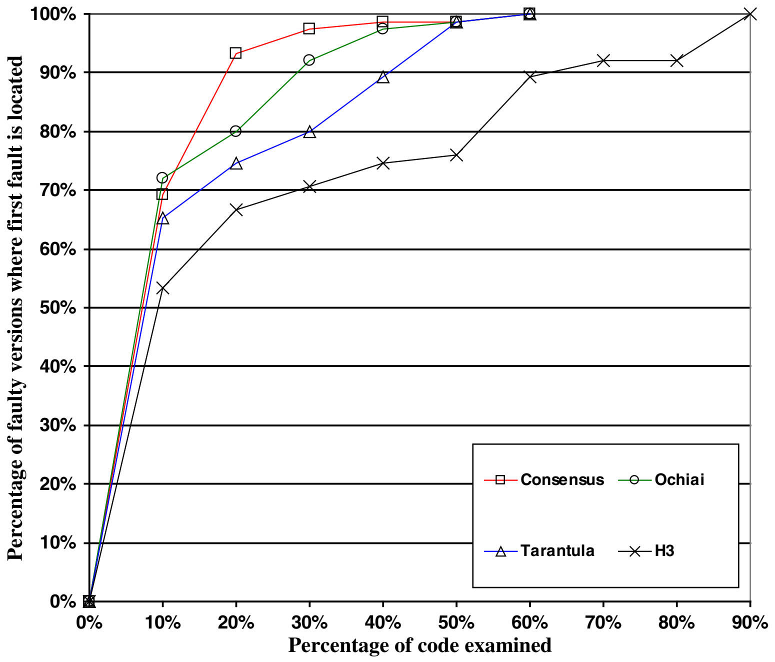


Figure 16. Effectiveness comparison for the 75 multifault versions of the Siemens suite using the *Expense* score.

Table VI. Total number of statements examined for the 75 multifault versions of the Siemens suite.

Tarantula	1618
Ochiai	1147
H3	3389
Consensus	1027

complete picture of effort spent, as programmers may not examine statements one at a time and may spend varying amounts of time examining different statements. Another threat would be how the rankings are consolidated together. Borda's method is a factor-5 approximation to the Kemeny optimal ranking [19] and may not satisfy all fairness criteria. Other approximations exist and have been reported in the literature. However, because this is a study to illustrate the value of consensus (rather than find the optimal consensus itself), their use has been deferred to future work. The use of a key (such as statement number) to break ties also represents a threat. However, for fairness the same key has been used across all the input rankings to consensus. Also the use of the statement number as a key offers no particular bias (in fact statement numbers are assigned by text editors/integrated development environments and not by us), and thus is not expected to have much of an impact on the fault localization results regardless of technique.

An important threat to the internal validity is the fact that some of the faulty versions (such as in the case of the grep, gzip, make, and Ant programs) have been created by seeding faults and using techniques such as mutation. However, this has only been done to increase the number of faulty versions we had to work with. Mutation-based fault injection has also been used in other studies [6, 24–26] where the authors state that such faults represent realistic faults and that the results obtained by using these faults are indeed reliable.

A threat to the external validity would be the choice of fault localization techniques and subject programs that have been used. A total of three different fault localization techniques are considered for consensus each different from the other in terms of its design. Because this is the first study of its kind (to the best of our knowledge), more fault localization techniques were not employed. This is because before the usefulness of the consensus-based strategy can be established, it may not make sense to use a large number of techniques. The programs of the Siemens suite are selected because they have served as a benchmark among fault localization studies. However, the Siemens suite consists of programs that are relatively small (in terms of the LOC count) and so to evaluate the proposed strategy on much larger programs, the grep, gzip, make, space, and Ant programs are also used. While we cannot and do not claim that our results can be generalized to all programs and techniques, the evaluation of three different techniques across six different sets of programs (12 programs in all) does allow us to have strong confidence in our results.

6. RELATED WORK

The ultimate goal of this paper is to present an approach to improve the quality of software fault localization (techniques) in a general sense. Driven by the same motivation, several fault localization techniques have been proposed, in addition to the techniques used for our experiments.

In [7], a nearest neighbor debugging technique was proposed by Renieres and Reiss that contrasts a failed test with another successful test (most similar to the failed one in terms of the 'distance' between them). If a bug is in the difference set between the failed execution and its most similar successful execution, it is located. For a bug not in the difference set, the technique continues by first constructing a program dependence graph and then checking adjacent unchecked nodes in the graph step by step until the bug is located. Also proposed in [7] are the set union and set intersection techniques. The former computes the set difference between the program spectrum of a failed test and the union spectra of a set of successful tests. It focuses on the source code that is executed by the failed test but not by any of the successful tests. The latter is based on the set difference between the intersection spectra of successful tests and the spectrum of the failed test. It focuses on statements that are executed by all successful tests but not by the failed test case.

Cleve and Zeller [4] report a program state-based debugging technique, cause transition, to identify the locations and times where a cause of failure changes from one variable to another. This is an extension of their earlier work with delta debugging [12, 30]. An algorithm named *cts* is proposed to quickly locate cause transitions in a program execution. A potential problem of the cause transition technique is that its cost is relatively high; there may exist thousands of states in a program execution, and delta debugging at each matching point requires additional test runs to narrow down the causes.

Liblit *et al.* proposed a statistical debugging technique (Liblit05) that can isolate bugs by instrumenting predicates at particular points [31]. Feedback reports are generated by these instrumented predicates. For each predicate P, the algorithm first computes *Failure*(P), the probability that P being true implies failure, and *Context*(P), the probability that the execution of P implies failure. Predicates that have *Failure*(P) – *Context*$(P) \leqslant 0$ are discarded. The remaining predicates are prioritized based on their '*importance*' scores, which indicate the relationship between predicates and program bugs. Predicates with a higher score are examined first to help programmers find bugs. Liu *et al.* proposed the SOBER technique to rank suspicious predicates [6]. First, $\pi(P)$, which is the probability that predicate P is evaluated to be true in each run, is computed as $n(t)/(n(t)+n(f))$, where $n(t)$ is the number of times P is evaluated to be true in a specific run and $n(f)$ is the number of times P is evaluated as false. If the distribution of $\pi(P)$ in failed runs is significantly different from that in successful runs, then P may be fault-relevant and this *relevance* is quantified by a ranking score. All instrumented predicates are then ranked in order of their scores and examined in order of fault-relevance.

Zhang *et al.* [32] presented a technique such that for a given failed test, they perform multiple executions against that test. In each execution, the outcome of one predicate is switched, and this process continues until the program produces the correct output as a result of the switch; then the corresponding predicate is a critical predicate. Bidirectional dynamic slices of such critical predicates are then computed to help programmers locate the bugs. There are also many slicing-based studies that can be further classified as static slicing-based [33, 34], dynamic slicing-based [35], and execution slicing-based [36] fault localization techniques. As opposed to directly assessing the suspiciousness of individual program entities, in [37], the authors instead focus on the propagation of infected program states.

In [38] a procedure is described to debug multiple faults present in a program simultaneously. First, 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 (Tarantula in the case of [38]) to locate the fault associated with that cluster. Consensus would apply here as well, as the only change to the procedure outlined would be to not use a single technique (such as Tarantula) alone to produce statement rankings, but rather to also include other high-quality techniques (such as the ones used in our experiments) to produce a more reliable, consensus-based ranking. The multiple-fault problem then boils down to a set of single-fault problems, for which the usefulness of the consensus-based strategy has already been demonstrated.

It is impossible to review all prominent fault localization techniques because of space limitations, and so we instead direct interested readers to [8] for a comprehensive survey on state-of-the-art fault localization techniques. To the best of our knowledge we are not aware of studies that recommend consensus similar to the manner that is proposed here. In [39] the authors evaluated the quality of fault localization with respect to multiple coverage types, namely, statements, branches, and data dependencies. Assigning suspiciousness to a statement involves taking into account statement coverage, branch coverage, and def-use coverage, and thus there is a flavor of consensus. However, this is different from our study as we propose consensus between different fault localization techniques and not different coverage types. Consensus, however, is no stranger to other related areas. Majority voting-based fault tolerance design techniques represent the first approach to hardware fault tolerance [40], and voting algorithms are often used to arbitrate between the results of several redundant modules in fault-tolerant systems. In highly dependable systems, voting is applied at many different levels and has also been used in many different areas such as distributed database systems, pattern recognition, object classification, handling heuristic knowledge, etc. [41]

N-version programming has also been proposed to incorporate fault tolerance into software [42]. Multiple versions of a program (N versions) are prepared and executed in parallel. Then their outputs are collected and examined, and if they are not identical then it is assumed that the majority is correct [43]. However, based on experiments conducted, the authors in [43] conclude that N-version programming must be used with care and that analysis of its reliability must include dependent errors. The discussion on N-version programming is quite relevant to the consensus-based strategy we have proposed. Essentially, each fault localization technique used, that is, one of the N-versions, is one realization of a common objective. Fault localization techniques can be based on the same set of intuitions, and therefore as per [43] they may suffer from similar problems thereby reducing the gain observed by consensus. However, as opposed to N-version programming where each version is implemented based on the same specification, the fault localization techniques used for consensus (in this paper) are not. In fact they can differ drastically in terms of their design and the way in which they utilize the input data to produce output rankings. Thus, our expectation (supported by data presented in this paper) is that the consensus of different fault localization techniques shall generally lead to better results.

Arrow's impossibility theorem [44], cited most often in the areas of social welfare and welfare economics, states that there is no consistent method that results in a fair choice among three or more candidates using preferential voting (a ranking of candidates in order of preference). More explicitly, the theorem considers certain conditions that are requirements of a fair voting scheme, and shows that if a decision-making body has at least two members and at least three options to decide among, then it is impossible to design a social welfare function that satisfies all the conditions simultaneously. Given that the number of statements in ranking is never expected to be as low as three, and we wish to incorporate several techniques into consensus, we deduce that the consensus-based ranking shall never completely be fair. However, this was not a requirement for us, and as is shown by our empirical data, quality results are achievable even if the ranking is not deemed fair by social welfare standards.

7. CONCLUSIONS AND FUTURE WORK

This paper has presented a strategy to improve the quality of software fault localization by making use of several different fault localization techniques via consensus as opposed to a singleton technique alone. Statement rankings (sorted in order of their likelihood of being faulty) that are produced by each technique are consolidated using the Borda method to produce a consolidated ranking. Data based on case studies that employ three fault localization techniques (Tarantula, Ochiai, and H3, across six different sets of programs), the programs of the Siemens suite, and the grep, gzip, make, space, and Ant programs, suggests that the consensus-based strategy provides close to the best performance (if not the best performance) irrespective of the data set. Furthermore, the proposed strategy has shown to be highly effective irrespective of whether a program has just one or multiple faults present in it.

The proposed strategy is highly extensible in that other fault localization techniques can be easily incorporated into the consensus, and it is also easily tractable in that removing a technique from consensus requires no more than just to discount its ranking. Also, all the fault localization techniques used as per our proposed strategy operate on the same input data, which means there is no additional data collection required to incorporate multiple fault localization techniques into the consensus. Furthermore, the proposed strategy only requires minimum overhead as (based on the experimental design described herein) the aggregate rankings can be computed in linear time.

Future work includes, but is not limited to, investigating the use of more accurate approximations to the Kemeny optimal ranking, further evaluating consensus using many different fault localization techniques, and conducting case studies on a broader set of subject programs. We also wish to investigate the possibility of introducing weighting schemes into Consensus such that each fault localization technique included in Consensus may be assigned a different weight, that is, one technique may be considered more important than another. Additionally, we would like to investigate whether the design-based consensus of some fault localization techniques may be achieved in an efficient manner. If this is the case, then we would also like to compare the effectiveness of the resultant technique with that of a ranking-based consensus.

REFERENCES

1. Vessey I. Expertise in debugging computer programs. *International Journal of Man-Machine Studies: A Process Analysis* 1985; **23**(5):459–494.
2. Abreu R, Zoeteweij P, Golsteijn R, van Gemund AJC. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 2009; **82**(11):1780–1792.
3. Agrawal H, Horgan JR, London S, Wong WE. Fault localization using execution slices and dataflow tests. *Proceedings of the Sixth IEEE International Symposium on Software Reliability Engineering*, October 1995; 143–151.
4. Cleve H, Zeller A. Locating causes of program failures. *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, Missouri USA, May 2005; 342–351.
5. Jones JA, Harrold MJ. Empirical evaluation of the Tarantula automatic fault-localization technique. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, California, USA, November 2005; 273–283.
6. Liu C, Fei L, Yan X, Han J, Midkiff SP. Statistical debugging: a hypothesis testing-based approach. *IEEE Transactions on Software Engineering* Oct. 2006; **32**(10):831–848.
7. Renieris M, Reiss SP. Fault localization with nearest neighbor queries. *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, Montreal, Canada, October 2003; 30–39.
8. Wong WE, Debroy V. A survey on software fault localization. *Technical Report UTDCS-45-09*, Department of Computer Science, University of Texas at Dallas, November 2009.
9. Wong WE, Qi Y, Zhao L, Cai KY. Effective fault localization using code coverage. *Proceedings of the 31st IEEE Computer, Software, and Applications Conference*, Beijing, China, July 2007; 449–456.
10. Wong WE, Wei T, Qi Y, Zhao L. A crosstab-based statistical method for effective fault localization. *Proceedings of the First International Conference on Software Testing, Verification and Validation*, Lillehammer, Norway, April 2008; 42–51.
11. Wong WE, Shi Y, Qi Y, Golden R. Using an RBF Neural Network to Locate Program Bugs. *Proceedings of the 19th IEEE International Symposium on Software Reliability Engineering*, Seattle, USA, November 2008; 27–38.
12. Zeller A. Isolating cause-effect chains from computer programs. *Proceedings of the 10th Symposium on Foundations of Software Engineering*, Charleston, South Carolina, November 2002; 1–10.
13. Dwork C, Kumar R, Naor M, Sivakumar D. Rank aggregation revisited, At: www.eecs.harvard.edu/∼michaelm/CS222/rank2.pdf.
14. Ailon N, Charikar M, Newman A. Aggregating inconsistent information: ranking and clustering. *Journal of the ACM* October 2008; **55**(5). Article 23.
15. Cranor LF. Declared-Strategy Voting: An Instrument for Group Decision-Making. *Ph. D. Thesis*, Washington University, 1996.
16. Conitzer V, Davenport A, Kalagnanam J. Improved bounds for computing Kemeny rankings. *Proceedings of the 21st National conference on AI*, Boston, Massachusetts, July 2006; 620–626.
17. Dwork C, Kumar R, Naor M, Sivakumar D. Rank aggregation methods for the web. *Proceedings of the 10th international conference on World Wide Web*, Hong Kong, May 2001; 613–622.
18. Borda JC. Mémoire sur les élections au scrutin, 1781. Histoire de l' Académie Royale des Sciences.
19. Coppersmith D, Fleischer L, Rudra A. Ordering by weighted number of wins gives a good ranking for weighted tournaments. *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, Miami, Florida, USA, January 2006; 776–782.
20. Jaccard P. The distribution of the flora in the alpine zone. *New Phytologist* February 1912; **11**(2):37–50.
21. Debroy V, Eric Wong W. On the consensus-based application of fault localization techniques. *Proceedings of the 2nd International Workshop on Program Debugging (IWPD)*, Munich, Germany, July 2011; 506–511.
22. http://www-static.cc.gatech.edu/aristotle/Tools/subjects/.
23. Software-artifact infrastructure repository. http://sir.unl.edu/portal/index.html.
24. Andrews JH, Briand LC, Labiche Y. Is mutation an appropriate tool for testing experiments? *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, Missouri, USA, May 2005; 402–411.
25. Do H, Rothermel G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering* Sept. 2006; **32**(9):733–752.
26. Namin AS, Andrews JH, Labiche Y. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* August 2006; **32**(8):608–624.
27. χSuds User's Manual, Telcordia Technologies, 1998.
28. http://www.atlassian.com/software/clover/(Clover:AcodecoverageanalysistoolforJava).
29. Yu Y, Jones JA, Harrold MJ. An empirical study on the effects of test-suite reduction on fault localization. *Proceedings of the International Conference on Software Engineering (ICSE)*, Leipzig, Germany, May 2008; 201–210.

30. Zeller A, Hildebrandt R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* February 2002; **28**(2):183–200.
31. Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI. Scalable statistical bug isolation. *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 2005; 15–26.
32. Zhang X, Gupta N, Gupta R. Locating faults through automated predicate switching. *Proceedings of the 28th International Conference on Software Engineering*, Shanghai, China, May 2006; 272–281.
33. Lyle JR, Weiser M. Automatic program bug location by program slicing. *Proceedings of the Second International. Conference on Computer and Applications*, Beijing, China, June 1987; 877–883.
34. Weiser M. Programmers use slices when debugging. *Communications of the ACM* July 1982; **25**(7):446–452.
35. Agrawal H, DeMillo RA, Spafford EH. Debugging with dynamic slicing and backtracking. *Software: Practice & Experience* June 1996; **23**(6):589–616.
36. Wong WE, Qi Y. Effective program debugging based on execution slices and inter-block data dependency. *Journal of Systems and Software* July 2006; **79**(7):891–903.
37. Zhang Z, Chan WK, Tse TH, Jiang B, Wang X. Capturing propagation of infected program states. *Proceedings of the 7th joint meeting of the European Software Engineering Conference (ESEC) and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Amsterdam, The Netherlands, August 2009; 43–52.
38. Jones JA, Bowring JM, Harrold J. Debugging in parallel. *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, London, UK, July 2007; 16–26.
39. Santelices R, Jones JA, Yu Y, Harrold MJ. Lightweight fault-localization using multiple coverage types. *Proceedings of the 31st International Conference on Software Engineering*, Canada, May 2009; 56–66.
40. von Neumann J. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies, Ann. of Math. Studies*, Shannon CE, McCarthy J (eds). Princeton University Press: Princeton, NJ, 1956; 43-98.
41. Shabgahi GL, Hirst AJ. A fuzzy voting scheme for hardware and software fault tolerance systems. *Fuzzy Sets and Systems* March 2005; **150**(3):579–598.
42. Chen L, Avizienis A. N-version Programming: A fault-tolerance approach to reliability of software operation. *Digest of Papers FTCS-8: Eight Annual International Conference on Fault Tolerant Computing*, Toulouse, France, June 1978; 3–9.
43. Knight JC, Leveson NG. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering* Jan. 1986; **12**(1):96–109.
44. Arrow KJ. A difficulty in the concept of social welfare. *The Journal of Political Economy* August 1950; **58**(4):328–346.