# Towards Better Fault Localization: A Crosstab-Based Statistical Approach

W. Eric Wong, Vidroha Debroy, *Student Member, IEEE*, and Dianxiang Xu

***Abstract*—It is becoming prohibitively expensive and time consuming, as well as tedious and error-prone, to perform debugging manually. Among the debugging activities, fault localization has been one of the most expensive, and therefore, a large number of fault-localization techniques have been proposed over the recent years. This paper presents a crosstab-based statistical technique that makes use of the coverage information of each executable statement and the execution result (success or failure) with respect to each test case to localize faults in an effective and efficient manner. A crosstab is constructed for each executable statement, and a statistic is computed to determine the suspiciousness of the corresponding statement. Statements with a higher suspiciousness are more likely to contain bugs and should be examined before those with a lower suspiciousness. Case studies are performed on both small- (the Siemens and Unix suites) and large-sized programs (space, grep, gzip, and make), and results suggest that the crosstab-based technique (CBT) is more effective (in terms of a smaller percentage of executable statements that have to be examined until the first statement containing the fault is reached) than other techniques, such as Tarantula. Further studies using the Siemens suite reveal that the proposed technique is also more effective at locating faults than other statistically oriented techniques, such as SOBER and Liblit05. Additional experiments evaluate the CBT from other perspectives, such as its efficiency in terms of time taken, its applicability to object-oriented languages (on a very large Java program: Ant), and its sensitivity to test suite size, and demonstrate its superior performance.**

***Index Terms*—Crosstab, debugging aids, software fault localization, statistical methods, testing and debugging.**

## I. Introduction

SOFTWARE today is larger and more complex than it has ever been, and software/program[1] debugging is becoming increasingly more difficult to perform. Fault localization is considered to be one of the most expensive program debugging activities [27], and manual fault localization has proven to be prohibitively time consuming and tedious. This has fueled the demand for fault-localization techniques that can automatically (or at least semiautomatically) lead programmers to the locations of faults in software, with minimum human intervention. To meet this demand, there has consequently been a surge in the development of fault-localization techniques [1], [2], [5], [14], [17]–[19], [24], [29]–[32], [37], [40], [41], each of which tries to address the fault-localization problem, effectively and efficiently, in its own way.

Several fault-localization techniques rely on a *dynamic analysis* of a program in the form of *program traces*, which are collected during program execution. The information from these traces can then be processed by various means in order to rank code based on its likelihood of containing bugs.[1] In [14], [30], etc., the authors make use of various heuristics and apply them to program traces (in the form of statement coverage) to rank statements[2] in order of their relative suspiciousness. This ranking can then be examined (in that same order), until a fault is found, thereby narrowing the programmer's search domain. However, such techniques are still based on experience or intuition, and must rely on personal judgment, which can be quite subjective in nature, as different people may not share the same intuitions. This is compounded by the fact that intuitions may be program dependent and may not generally apply everywhere. Along more objective lines, several statistically based techniques do exist, such as the ones proposed in [16]–[18], etc., which rely on the instrumentations and evaluations of predicates in programs to produce a ranking of suspicious predicates, which can be examined in order to find faults. However, these techniques are constrained by the overhead involved in the instrumentation and sampling of predicates.

In light of such limitations, we propose a cross tabulation (hereafter referred to simply as "crosstab") analysis-based technique for fault localization. Similar to other heuristics-based techniques, our crosstab-based technique (hereafter referred to as CBT) also uses the coverage information of each statement and the execution result (success or failure) with respect to each test case. However, it is different from the heuristic-based techniques, in that it is based on a well-defined statistical analysis, whereas the heuristic-based techniques are not. At the same time, since CBT only employs the coverage information of each statement, it does not require any additional instrumentation of code to evaluate predicate outcomes, etc. In our technique, a crosstab is constructed for each statement and a statistic is computed to determine the suspiciousness of the corresponding statement. All the statements can, therefore, be ranked in descending order of their suspiciousness. Statements toward the top have a higher suspiciousness than those toward

Manuscript received August 27, 2010; revised December 15, 2010; accepted January 22, 2011. Date of publication May 23, 2011; date of current version April 11, 2012. This work was supported in part by the National Science Foundation under Grant DUE-1023071. This paper was recommended by Associate Editor J. Wang.

W. E. Wong and V. Debroy are with the Department of Computer Science, University of Texas at Dallas, Dallas, TX 75219 USA (e-mail: wong@utdallas.edu; vxd024000@utdallas.edu).

D. Xu is with the Department of Computer Science, Dakota State University, Madison, SD 57042 USA (e-mail: dianxiang.xu@dsu.edu)

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSMCC.2011.2118751

[1] In this paper, "fault" and "bug" are used interchangeably. In addition, "program" and "software" are also used interchangeably.

[2] Only executable statements are considered. See Section III-C for more details. Hereafter, we refer to "executable statements" simply as "statements."

the bottom and, correspondingly, a higher likelihood of containing bugs. Statements are examined top–down until a bug-containing statement is found.

The underlying assumption is that a programmer can and will correctly identify a faulty statement when examining one (see Section VIII-E for more details). Should a bug span multiple statements, then the examination stops once the first statement containing the bug is reached. This is because our objective is not to find all the statements that have to be fixed for a given bug, but just to find one of these faulty statements as the starting point to initiate the bug-fixing process. In no way does this imply that techniques, such as CBT, are only limited to faults that span one statement. Note that multiline faults are different from multiple faults present in the same program (which are discussed in Section VII). By a fault that spans multiple statements, we mean that a single fault might correspond to multiple locations in the code. Consider the case, where the clauses of an "if–else" construct are accidentally swapped. In this case, the fault manifests itself in the "if" portion of the construct as well as in the "else" portion of the construct, i.e., two locations corresponding to the same fault.

The effectiveness of CBT for fault localization is evaluated across several case studies: two suites of small programs (seven in the Siemens [26] and ten in the Unix suites [28]), and four relatively larger programs (space, grep, gzip, and make) [10] in the C language and a very large program in Java (Ant [10]). The effectiveness is also compared to that of Tarantula [14] as the latter is popular and well known, and has already been shown to be more effective than several other fault-localization techniques, such as set union, set intersection, nearest neighbor, and cause transitions on the Siemens suite [14]. The criteria for the comparison between two fault-localization techniques are described in detail in Section III-C. A comparison of fault-localization effectiveness between CBT and other statistically oriented fault-localization techniques, such as Liblit05 [17] and SOBER [18], is conducted as well on the Siemens suite. Studies are also performed to evaluate CBT from other perspectives, such as its efficiency in terms of time taken and its sensitivity to test suite size. In addition, experiments demonstrate how the crosstab-based fault-localization technique may easily be applied to programs with multiple faults are also included.

The remainder of this paper is organized as follows: Section II gives an overview of work related to what is presented in this paper. Section III explains our methodology in detail along with an illustrative example and the main criterion for the comparison of fault-localization techniques. Section IV presents the results of empirical studies conducted to observe the application of our methodology to various programs. A comparison of the effectiveness between CBT and Tarantula is also included. Section V provides a comparison between CBT and two other statistical fault-localization techniques—SOBER and Liblit05. This is followed by Section VI, which explores other criteria to evaluate the effectiveness of CBT, which in turn is followed by a discussion on fault localization for programs with multiple bugs in Section VII. Section VIII contains further discussion and the threats to the validity of the approach and results presented in this paper. Our conclusions and directions for future research appear in Section IX.

## II. Related Work

Several different studies have proposed, and reported results on, various fault-localization techniques. Similar to ours, Tarantula is a fault-localization technique [14], which also makes use of coverage and execution result information to compute the suspiciousness of each statement, as shown in (8) (see Section VIII-C). Renieris and Reiss [24] propose a nearest neighbor debugging technique that contrasts a failed test with another successful test, which is most similar to the failed one in terms of the "distance" between them. The execution of a test is represented as a sequence of basic blocks that are sorted by their execution times. If a bug is in the difference set between the failed execution and its most similar successful execution, it is located. For a bug that is not contained in the difference set, the technique continues by first constructing a program dependence graph, and then, including and checking adjacent unchecked nodes in the graph step by step until the bug is located. The set union and set intersection techniques are also reported in [24]. The former computes the set difference between the "program spectra" 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 spectra of the failed test. It focuses on statements that are executed by every successful test but not the failed test case.

In [5], Cleve and Zeller 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 [37], [38]. 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.

Zhang *et al*. [40] present a technique such that for a given failed test, their technique requires 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; this predicate is a critical predicate. Bidirectional dynamic slices of such critical predicates are then computed to help programmers to locate the bugs.

Liblit *et al*. propose a statistical debugging technique (Liblit05) that can isolate bugs by instrumenting predicates at particular points [17]. 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 $Failure(P) - Context(P) \leq 0$ 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*. propose the SOBER technique to rank suspicious predicates [18]. 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. A comparison among Liblit05, SOBER, and CBT is provided in Section V.

As opposed to directly assessing the suspiciousness of individual program entities, Zhang *et al*. instead focus on the propagation of infected program states among the entities in [39]. In [25], the original Tarantula technique [14] is extended by making use of the Ochiai coefficient [1] to evaluate the quality of fault localization with respect to multiple coverage types, namely, statements, branches, and data dependencies. Their study shows that no single coverage type performs best for all studied faults, and that different kinds of faults are best localized by different coverage types. A new coverage-based approach to fault localization that leverages each coverage type via combination is also presented in [25], and because data dependencies can be expensive to monitor, the effects of replacing data dependence coverage with an approximation inferred from branch coverage are also investigated. For the purposes of the CBT technique presented in this paper, only statement-based coverage is considered to be an input to CBT, and the effects of using multiple coverage types with CBT is deferred to future study.

Wong *et al*. propose a set of coverage-based heuristics for fault localization in [30] and address two issues: first, how can each additional failed test case aid in locating program faults, and second, how can each additional successful test case help in locating program faults. They propose that with respect to a piece of code, the contribution of the first failed test case that executes it in computing its likelihood of containing a fault is larger than or equal to that of the second failed test case that executes it, which in turn is larger than or equal to the third failed test case, etc. This principle is also applied to the contribution provided by the successful tests.

## III. Crosstab-Based Technique

Having summarized the state of the art in the area of fault localization, we now present our proposed methodology. We first describe CBT in detail, followed by an example to walk the reader through how CBT is to be applied. Then, we present the primary criterion used to evaluate the effectiveness of the technique, which also serves as a means to compare various fault-localization techniques to one another. Other evaluation criteria are also presented and discussed in Section VI.

TABLE I
Notations Used in This Paper

| | |
|---|---|
| $N$ | total number of test cases |
| $N_F$ | total number of failed test cases |
| $N_S$ | total number of successful test cases |
| $N_C(\omega)$ | number of test cases covering[a] $\omega$ |
| $N_{CF}(\omega)$ | number of failed test cases covering $\omega$ |
| $N_{CS}(\omega)$ | number of successful test cases covering $\omega$ |
| $N_U(\omega)$ | number of test cases not covering $\omega$ |
| $N_{UF}(\omega)$ | number of failed test cases not covering $\omega$ |
| $N_{US}(\omega)$ | number of successful test cases not executing $\omega$ |

[a] "A statement is *covered* by a test case" and "a statement is *executed* by a test case" are used interchangeably.

TABLE II
Crosstab for Each Statement

| | $\omega$ is covered | $\omega$ is not covered | $\sum$ |
|---|---|---|---|
| successful executions | $N_{CS}(\omega)$ | $N_{US}(\omega)$ | $N_S$ |
| failed executions | $N_{CF}(\omega)$ | $N_{UF}(\omega)$ | $N_F$ |
| $\sum$ | $N_C(\omega)$ | $N_U(\omega)$ | $N$ |

### A. Overview

Consider a statement $\omega$ in the program that is being debugged. We define the following notations in order to facilitate further discussion regarding the CBT (see Table I).

The crosstab analysis is used to study the relationship between two or more categorical variables [7]–[9]. A crosstab, as shown in Table II, is constructed for each statement such that the crosstab has two columnwise categorical variables—*covered* and *not covered*; and two rowwise categorical variables—*successful* and *failed*.

For each crosstab so constructed, we now conduct a hypothesis test[3] to check for a dependence relationship. The null hypothesis is "*program execution result is independent of the coverage of statement* $\omega$."

A chi-square test can determine whether this hypothesis should be rejected. The chi-square statistic is given by

$$\chi^2(\omega) = \frac{(N_{\mathrm{CF}}(\omega) - E_{\mathrm{CF}}(\omega))^2}{E_{\mathrm{CF}}(\omega)} + \frac{(N_{\mathrm{CS}}(\omega) - E_{\mathrm{CS}}(\omega))^2}{E_{\mathrm{CS}}(\omega)} + \frac{(N_{\mathrm{UF}}(\omega) - E_{\mathrm{UF}}(\omega))^2}{E_{\mathrm{UF}}(\omega)} + \frac{(N_{\mathrm{US}}(\omega) - E_{\mathrm{US}}(\omega))^2}{E_{\mathrm{US}}(\omega)} \tag{1}$$

where $E_{\mathrm{CF}}(\omega) = N_C(\omega) \times N_F/N$, $E_{\mathrm{CS}}(\omega) = N_C(\omega) \times N_S/N$, $E_{\mathrm{UF}}(\omega) = N_U(\omega) \times N_F/N$, and $E_{\mathrm{US}}(\omega) = N_U(\omega) \times N_S/N$. The intuition behind (1) and its relevance to fault localization is explained in detail in Section VIII-C. Under the null hypothesis, the statistic $\chi^2(\omega)$ has an approximately chi-square distribution [7]. Given a level of significance $\sigma$, we can find the corresponding chi-square critical value $\chi^2_\sigma$ from the chi-square distribution table. If $\chi^2(\omega) > \chi^2_\sigma$, we reject

[3]There is a significant difference between the way a hypothesis test is traditionally applied and the way it is made use of for our purposes. Instead of the "dependence"/"independence" relationship, we are more interested in the "degree of association" between the execution result and the coverage of each statement. As a result, the suspiciousness of each statement does not depend on whether the hypothesis test is accepted or not. See (2)–(4), as well as Section VIII-A.

the null hypothesis, i.e., the execution result depends on the coverage of $\omega$. Otherwise, we accept the null hypothesis, i.e., the execution result and the coverage of $\omega$ are "independent." Note that the "dependence" relationship indicates a high association between the variables, whereas the "independence" relationship implies a low association. Instead of the so-called "dependence"/"independence" relationship, we are more interested in the degree of association between the execution result and the coverage of each statement. This degree can be measured based on the standard chi-square statistic. However, such a measure increases with increasing sample size. As a result, the measure by itself may not give the "true" degree of association. One way to address this is to use the *coefficient of contingency* computed as follows:

$$\mathcal{M}(\omega) = \frac{\chi^2(\omega)/N}{\sqrt{(row-1)(col-1)}} \tag{2}$$

where *row* and *col* are the number of categorical variables in all rows and columns, respectively, of the crosstab [7]. This coefficient lies between 0 and 1. When $\chi^2(\omega) = 0$, $\mathcal{M}(\omega)$ has the lower limit 0 for complete independence. In the case of complete association, the coefficient $\mathcal{M}(\omega)$ can reach the upper limit 1 when $row = col$. In addition, a larger coefficient implies a higher association between the execution result and the coverage of $\omega$ [7], [15]. From (2), if $N$, *row*, and *col* are fixed (which is true in our case as $row = col = 2$), then $\mathcal{M}(\omega)$ increases with increasing $\chi^2(\omega)$. Under this condition, the chi-square statistic $\chi^2(\omega)$ for statement $\omega$ gives a good indication of the degree of the association between the execution result and the coverage of $\omega$.

Next, we need to decide whether it is the failed or the successful execution result that is more associated with the coverage of the statement. For each statement $\omega$, we compute $\mathcal{P}_F(\omega)$ and $\mathcal{P}_S(\omega)$ as $N_{\mathrm{CF}}(\omega)/N_F$ and $N_{\mathrm{CS}}(\omega)/N_S$, which are the fractions of all failed and successful tests that execute $\omega$. From Table II and [7] and [15], if we have $N_{\mathrm{CF}}(\omega) \times N_{\mathrm{US}}(\omega) > N_{\mathrm{CS}}(\omega) \times N_{\mathrm{UF}}(\omega)$, then the coverage of $\omega$ is positively associated with the failed execution. This implies that if $\mathcal{P}_F(\omega)$ is larger than $\mathcal{P}_S(\omega)$, then the association between the failed execution and the coverage of $\omega$ is higher than that between the successful execution and the coverage of $\omega$. Let us define a statistic

$$\varphi(\omega) = \frac{\mathcal{P}_F(\omega)}{\mathcal{P}_S(\omega)} = \frac{N_{\mathrm{CF}}(\omega)/N_F}{N_{\mathrm{CS}}(\omega)/N_S}. \tag{3}$$

If $\varphi(\omega) = 1$, we have $\chi^2(\omega) = 0$, which implies the execution result is completely independent of the coverage of $\omega$. In this case, we say the coverage of $\omega$ makes the same contribution to both the failed and the successful execution result. If $\varphi(\omega) > 1$, the coverage of $\omega$ is more associated with the failed execution. Otherwise, the coverage of $\omega$ is more associated with the successful execution. Depending on the values of $\chi^2(\omega)$ and $\varphi(\omega)$, statements of the program being debugged can be classified into one of the following classes.

1) Statements with $\varphi > 1$ and $\chi^2 > \chi^2_\sigma$ have a high degree of association between their coverage and the failed execution result.
2) Statements with $\varphi > 1$ and $\chi^2 \leq \chi^2_\sigma$ have a low degree of association between their coverage and the failed execution result.
3) Statements with $\varphi < 1$ and $\chi^2 > \chi^2_\sigma$ have a high degree of association between their coverage and the successful execution result.
4) Statements with $\varphi < 1$ and $\chi^2 \leq \chi^2_\sigma$ have a low degree of association between their coverage and the successful execution result.
5) Statements with $\varphi = 1$ (under this situation $0 = \chi^2 < \chi^2_\sigma$), whose coverage is independent of the execution result.

Statements in the first class are most likely to contain program bugs (i.e., have the highest suspiciousness) followed by those in the second, the fifth, and the fourth classes, respectively. Statements in the third class are least likely to contain bugs (i.e., have the least suspiciousness). As discussed earlier, the larger the coefficient $\mathcal{M}(\omega)$, the higher the association between the execution result and the coverage of $\omega$. Hence, for statements in the first and the second classes, those with a larger $\mathcal{M}$ are more suspicious. On the other hand, for statements in the third and the fourth classes, those with a smaller $\mathcal{M}$ are more suspicious. Formally, the suspiciousness of a statement $\omega$ can now be defined by a statistic $\zeta$ as follows:

$$\zeta(\omega) = \begin{cases} \mathcal{M}(\omega), & \text{if } \varphi(\omega) > 1 \\ 0, & \text{if } \varphi(\omega) = 1 \\ -\mathcal{M}(\omega), & \text{if } \varphi(\omega) < 1. \end{cases} \tag{4}$$

The larger the value of $\zeta$, the more suspicious the statement $\omega$. Having formally defined the suspiciousness function, we make additional observations prior to proceeding. First, since for our purposes, we know that the values of *row* and *col* in (2) are both always 2, (2) simplifies to

$$\mathcal{M}(\omega) = \frac{\chi^2(\omega)}{N}. \tag{5}$$

Second, we note that in our context, we are not as concerned with the absolute suspiciousness assigned to a statement, as much as we are with how statements are assigned suspiciousness in a more relativistic sense, i.e., the suspiciousness of one statement relative to the other. Assuming a ranking of statements is generated based on assigned suspiciousness, then dividing the assigned suspiciousness of each statement by the same constant (with the exception of things like zero and infinity) does not affect the ranking at all. The position of each statement with respect to the other is maintained. Thus, the normalization of the chi-square statistic in our context is immaterial (other than the fact that it squashes the assigned suspiciousness between 0 and 1) vis-à-vis the final ranking, and the corresponding fault-localization results. Thus, the normalization term $N$ in (5) can be safely dropped, eliminating the need for the coefficient of contingency $\mathcal{M}(\omega)$ and essentially reducing (4) to

$$\zeta(\omega) = \begin{cases} \chi^2(\omega), & \text{if } \varphi(\omega) > 1 \\ 0, & \text{if } \varphi(\omega) = 1 \\ -\chi^2(\omega), & \text{if } \varphi(\omega) < 1. \end{cases} \tag{6}$$

TABLE III
COVERAGE AND EXECUTION RESULTS OF THE EXAMPLE IN SECTION III-B

| | $s_1$ | $s_2$ | $s_3$ | $s_4$ | $s_5$ | $s_6$ | $s_7$ | $s_8$ | $s_9$ | $s_{10}$ | $r$ | | $s_1$ | $s_2$ | $s_3$ | $s_4$ | $s_5$ | $s_6$ | $s_7$ | $s_8$ | $s_9$ | $s_{10}$ | $r$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $t_1$ | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | $t_{19}$ | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| $t_2$ | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | $t_{20}$ | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| $t_3$ | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | $t_{21}$ | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| $t_4$ | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | $t_{22}$ | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| $t_5$ | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | $t_{23}$ | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| $t_6$ | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | $t_{24}$ | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| $t_7$ | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | $t_{25}$ | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| $t_8$ | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | $t_{26}$ | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| $t_9$ | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | $t_{27}$ | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| $t_{10}$ | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | $t_{28}$ | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| $t_{11}$ | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | $t_{29}$ | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| $t_{12}$ | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | $t_{30}$ | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| $t_{13}$ | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | $t_{31}$ | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| $t_{14}$ | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | $t_{32}$ | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 0 |
| $t_{15}$ | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | $t_{33}$ | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| $t_{16}$ | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | $t_{34}$ | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 |
| $t_{17}$ | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | $t_{35}$ | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 |
| $t_{18}$ | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | $t_{36}$ | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |

Thus, it is (6) and not (2) and (4), which is finally used for the purposes of CBT. However, in situations where normalization is a must, (2) in conjunction with (4) may equivalently be used in place of (6) without altering the fault-localization results.

### B. Illustrative Example

We now use a numerical example to explain how CBT can be used to compute the suspiciousness of each statement. Without resorting to complex notation, we assume Table III gives the statement coverage and execution results of the program being debugged. For example, the column labeled with $s_1$ shows how the statement $s_1$ is covered with respect to each test case. An entry 1 implies $s_1$ is covered by the corresponding test and an entry 0 means it is not. The column labeled with $r$ indicates whether the program execution succeeds or fails with respect to the corresponding test case. An entry 1 implies a failed execution, and an entry 0 means a successful execution. Of the 36 test cases, there are nine failed tests (e.g., $t_1$) and 27 successful tests (e.g., $t_2$).

We then construct the crosstab for $s_1$, as shown in Table IV. We have $E_{\mathrm{CF}}(s_1) = N_C(s_1) \times N_F/N = 25 \times 9/36 = 6.25$, $E_{\mathrm{CS}}(s_1) = N_C(s_1) \times N_S/N = 25 \times 27/36 = 18.75$, $E_{\mathrm{UF}}(s_1) = N_U(s_1) \times N_F/N = 11 \times 9/36 = 2.75$, and $E_{\mathrm{US}}(s_1) = N_U(s_1) \times N_S/N = 11 \times 27/36 = 8.25$. Using (1), the chi-square test statistic for $s_1$ is as follows:

$$\begin{aligned}\chi^2(s_1) &= \frac{(N_{\mathrm{CF}}(s_1) - E_{\mathrm{CF}}(s_1))^2}{E_{\mathrm{CF}}(s_1)} + \frac{(N_{\mathrm{CS}}(s_1) - E_{\mathrm{CS}}(s_1))^2}{E_{\mathrm{CS}}(s_1)} \\ &\quad + \frac{(N_{\mathrm{UF}}(s_1) - E_{\mathrm{UF}}(s_1))^2}{E_{\mathrm{UF}}(s_1)} + \frac{(N_{\mathrm{US}}(s_1) - E_{\mathrm{US}}(s_1))^2}{E_{\mathrm{US}}(s_1)} \\ &= \frac{(9-6.25)^2}{6.25} + \frac{(16-18.75)^2}{18.75} + \frac{(0-2.75)^2}{2.75} \\ &\quad + \frac{(11-8.25)^2}{8.25} = 5.2800.\end{aligned}$$

If we choose the level of significance as 0.05, the chi-square critical value is 3.841. Since $\chi^2(s_1)$ is larger than 3.841, the null hypothesis for $s_1$ should be rejected. Similarly, we can compute $\chi^2$ for other statements. For example, we have $\chi^2(s_2) = 4.4954$, $\chi^2(s_3) = 0.1481$, and $\chi^2(s_4) = 1.3333$. Together, we conclude the execution result has a higher association with the coverage of $s_1$ and $s_2$, but a lower association with $s_3$ and $s_4$. From Table IV, we compute $\mathcal{P}_F(s_1) = 100\%$ and $\mathcal{P}_S(s_1) = 59.26\%$, which give $\varphi(s_1) = 1.6875$ [see (3)]. For statements $s_2$, $s_3$, and $s_4$, their $\varphi$ values are $\varphi(s_2) = 0.3529$, $\varphi(s_3) = 0.8571$, and $\varphi(s_4) = 0.6000$. Finally, from (6), we have $\zeta(s_1) = \chi^2(s_1) = 5.28$, $\zeta(s_2) = -\chi^2(s_2) = -4.4954$, $\zeta(s_3) = -\chi^2(s_3) = -0.1481$, and $\zeta(s_4) = -\chi^2(s_4) = -1.333$.

Table V lists the values of $\chi^2$, $\varphi$, and $\zeta$ for all the statements in Table III. Sorted in descending order of suspiciousness, statement $s_8$ should be examined first to locate bugs, followed by $s_1$, $s_5$, $s_{10}$, $s_9$, $s_6$, $s_3$, $s_7$, $s_4$, and $s_2$.

TABLE IV
CROSSTAB FOR STATEMENT $s_1$

| | $s_1$ is covered | $s_1$ is not covered | $\sum$ |
|---|---|---|---|
| successful executions | 16 | 11 | 27 |
| failed executions | 9 | 0 | 9 |
| $\sum$ | 25 | 11 | 36 |

TABLE V
STATISTICS FOR STATEMENTS IN TABLE III

| | $\chi^2$ | $\varphi$ | $\zeta$ |
|---|---|---|---|
| $s_1$ | 5.2800 | 1.6875 | 5.28 |
| $s_2$ | 4.4954 | 0.3529 | –4.4954 |
| $s_3$ | 0.1481 | 0.8571 | –0.1481 |
| $s_4$ | 1.3333 | 0.6000 | –1.333 |
| $s_5$ | 1.8204 | 1.6364 | 1.8204 |
| $s_6$ | 0.1558 | 1.2000 | 0.1558 |
| $s_7$ | 0.6000 | 0.7500 | –0.6 |
| $s_8$ | 7.6364 | 2.0769 | 7.6364 |
| $s_9$ | 0.1846 | 1.1053 | 0.1846 |
| $s_{10}$ | 1.3333 | 1.5000 | 1.333 |

### C. Evaluation Criteria

Renieris and Reiss [24] assign a score to every faulty version of each 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. This score or effectiveness measure is later adopted by Cleve and Zeller [5] and is defined as $1 - (|N|/|\mathrm{PDG}|)$, where $N$ is the set of all nodes examined and PDG is the set of all nodes in the program dependence graph. Instead of the program dependence graph, the Tarantula fault localizer directly uses the program's source code. Therefore, in order to make their effectiveness computations comparable with those of the program dependence graph, Jones and Harrold [14] consider only executable statements to compute their score. They omit from consideration source code, such as blank lines, comments, function, and variable declarations. They also combine multiline statements into one source code line so that they are only counted once. Thus, the comparison is now a fair one—only statements that can be represented in the program dependence graph are considered [14]. Since the crosstab fault localizer also operates directly on the program source, we follow the same strategy and consider only executable statements in all of our experiments.

However, while Jones and Harrold [14] define their score to be the percentage of code that need not be examined to find a fault, we feel it is more straightforward to present the percentage of code that *has to be examined* to find a fault. This modified score is hereafter referred to as $\mathcal{EXAM}$ and is defined as the percentage of executable statements that have to be examined until the first statement containing the bug is reached. We note that the two scores are equivalent and it is easy to derive one from the other. A similar such modification is made by Liu *et al*. [18]; they define their effectiveness ($T$-score) as $T = (|V_{\mathrm{examined}}|/|V|) \times 100\%$, where $|V|$ is the size of the program dependence graph and $|\mathrm{V_{examined}}|$ is the number of statements examined in a breadth first search before a faulty node is reached.

Thus, the effectiveness of different fault-localization techniques can be compared based on $\mathcal{EXAM}$. For a faulty version $\mathcal{U}$,[4] if the $\mathcal{EXAM}$ assigned to $\mathcal{U}$ by technique $\alpha$ is smaller than that assigned by technique $\beta$ (i.e., technique $\alpha$ can guide the programmer to the fault in $\mathcal{U}$ by examining less code than technique $\beta$), then $\alpha$ is more effective than $\beta$ to locate the bug in $\mathcal{U}$. Stated differently, we can also say $\alpha$ performs better than $\beta$. In the case of multiple faulty versions, the $\mathcal{EXAM}$ score can still be utilized by comparing the number of faulty versions on which one technique yields a smaller $\mathcal{EXAM}$ score than another. While the $\mathcal{EXAM}$ score is the primary measure of effectiveness for the technique presented; other criteria have also been taken into consideration and these are further discussed in Section VI.

## IV. Empirical Studies

So far, the focus of this paper has been to distinguish CBT from other fault-localization techniques that might be based on intuition, with respect to the statistical foundation of CBT. In this section, we also empirically compare CBT with Tarantula on six sets of programs—the *Siemens suite*, the *Unix suite*, *space*, *grep*, *gzip*, and *make*.

Note that both Tarantula and CBT produce ranked lists of statements that have been sorted in order of their suspiciousness and it is possible that the same suspiciousness value may be assigned to multiple statements. Let us assume a bug-containing statement and several correct statements share the same suspiciousness. Then, in the best case, we examine the bug-containing statement first, and in the worst case, we examine it last and have to examine many correct statements before we discover the bug. This results in two different levels of effectiveness—the "best" and the "worst." In all our experiments, we assume that for the "best" effectiveness, we examine the faulty statement first, and for the "worst" effectiveness, we examine the faulty statement last. Data are presented for each of these two levels of effectiveness. Hereafter, we refer to the best effectiveness of the Tarantula technique as TBest and the worst effectiveness as TWorst. We follow this naming convention for CBT as well, which denotes the best crosstab-based effectiveness as CBest and the worst crosstab-based effectiveness as CWorst.

### A. Data Collection

For the Siemens suite, Unix suite, and space, all executions are on a PC with a 2.13- GHz Intel Core 2 Duo CPU and 8 GB physical memory. The operating system is SunOS 5.10 (Solaris 10) and the compiler used is gcc 3.4.3. For grep, gzip, and make, the executions are on a Sun-Fire-280R machine with SunOS 5.10 as the operating system and gcc 3.4.4 as the compiler. Each faulty version is executed against all its corresponding available test cases. When comparing the outputs of the faulty and correct versions of a program, any deviation from the correct output is recorded as a failure, and a success is recorded if the test case output is identical between the correct and faulty versions. To collect coverage information, we instrument the programs using a revised version of $\chi$Suds [35], which has the ability to record which statements of the source code are executed by a test case and how many times each of the statements is executed. In order for a statement to have been covered by a test case, we require that it must have been executed by the test case at least once.

### B. Programs of a Relatively Smaller Size

Each program in this category has an LOC (*lines of code*—the size of the program before any nonexecutable code is removed) count of less than 1000, and therefore, is identified as "a program of a relatively smaller size."

*1) Siemens Suite:* The programs of the Siemens suite have been employed by many fault localization related studies in their experiments (such as [5], [14], [18], and [34]). All the correct and faulty versions of the programs (written in the C language) and test cases are downloaded from [26]. Table VI gives a summary of this suite that includes the name of each program, the number of faulty versions, LOC, the number of executable statements, and the number of test cases. Further details on the Siemens suite can be found in [12].

[4]In our study, each faulty version is executed against all the corresponding available test cases.

TABLE VI
SUMMARY OF THE SIEMENS SUITE

| Program | Number of faulty versions[a] | LOC | Number of executable statements | Number of test cases |
|---|---|---|---|---|
| print_tokens | 7 | 565 | 175 | 4130 |
| print_tokens2 | 10 | 510 | 178 | 4115 |
| replace | 32 | 563 | 216 | 5542 |
| schedule | 9 | 412 | 121 | 2650 |
| schedule2 | 10 | 307 | 112 | 2710 |
| tcas | 41 | 173 | 55 | 1608 |
| tot_info | 23 | 406 | 113 | 1052 |

[a]Three of the 132 faulty versions not used.

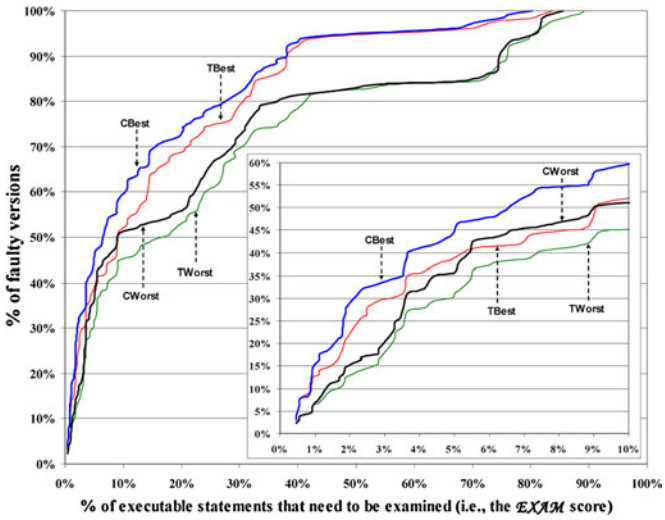


Fig. 1. Effectiveness comparison on the Siemens suite.

Of the 132 faulty versions, three are not used in our study because 1) none of the test cases downloaded can reveal the fault in version 9 of "schedule2," and 2) the faults in versions 4 and 6 of "print_tokens" are in the header files instead of in the C files. Compared with a previous study [14], ten faulty versions are excluded. Since the number of faulty versions and the number of test cases in our study are slightly different from those in [14], it is necessary to recalculate the effectiveness of Tarantula using our test data and their ranking mechanism [(8) as presented in Section VIII-C] to arrive at a fair comparison.

Fig. 1 gives the effectiveness of the Tarantula and the CBT. For a given $x$ value, its corresponding $y$ value is the cumulative percentage of the faulty versions, whose $\mathcal{EXAM}$ score is less than or equal to $x$. The two curves CBest and CWorst give the best and worst effectiveness of CBT, and the curves labeled TBest and TWorst give the best and worst effectiveness of Tarantula, respectively. Based on the figure, we observe that not just is CBest better than TBest, but CWorst is also better than TWorst. For this figure and for those that follow, zoom-ins are provided as an enhanced perspective of the range of $\mathcal{EXAM}$ scores from 0% to 10%. In addition, since the curves are displayed in different colors, it is better to view these figures in color.

Data in Table VII provides a pairwise comparison between the effectiveness of CBT and Tarantula to illustrate on how many faulty versions one technique performs better than the other, the same as the other, and worse than the other.[5] For example, we observe that CBest is better than TBest on 51 of the 129 faulty versions and is equal to TBest on 76 of the faulty versions. CBest performs worse than TBest on only two versions. Data has also been provided to show the pairwise comparison between CWorst and TWorst, as well as between CWorst and TBest. We observe that CWorst only performs worse than TWorst on five versions. On the other hand, CWorst performs better than TBest on 30 of the 129 faulty versions.

TABLE VII
PAIRWISE COMPARISON ON THE SIEMENS SUITE

| | CBest versus TBest | CWorst versus TWorst | CWorst versus TBest |
|---|---|---|---|
| More effective | 39.53%(51) | 39.53%(51) | 23.26%(30) |
| Same effectiveness | 58.92%(76) | 56.59%(73) | 12.40%(16) |
| Less effective | 1.55%(2) | 3.88%(5) | 64.34%(83) |

TABLE VIII
TOTAL NUMBER OF STATEMENTS EXAMINED FOR ALL THE 129 FAULTY VERSIONS OF THE SIEMENS SUITE

| CBest | CWorst | TBest | TWorst |
|---|---|---|---|
| 2005 | 2873 | 2453 | 3311 |

TABLE IX
SUMMARY OF THE UNIX SUITE

| Program | Number of faulty versions | LOC | Number of executable statements | Number of test cases |
|---|---|---|---|---|
| Cal | 20 | 202 | 88 | 162 |
| Checkeq | 20 | 102 | 57 | 166 |
| Col | 30 | 308 | 165 | 156 |
| Comm | 12 | 167 | 76 | 186 |
| Crypt | 14 | 134 | 77 | 156 |
| Look | 14 | 170 | 70 | 193 |
| Sort | 21 | 913 | 448 | 997 |
| Spline | 13 | 338 | 126 | 700 |
| Tr | 11 | 137 | 81 | 870 |
| Uniq | 17 | 143 | 71 | 431 |

Table VIII summarizes the comparison in terms of the total number of statements that need to be examined in order to locate all of the faults studied. We observe that CBest leads to the discovery of all of the faults by examining a fewer number of statements than TBest. The same is true when we compare CWorst and TWorst. An important point is that the statements examined by CBest may not be a subset of those examined by TBest. The same applies between CWorst and TWorst.

Taking the three modes of comparison (see Fig. 1, and Tables VII and VIII) into account, we conclude that CBT performs better than Tarantula on the Siemens suite.

*2) Unix Suite:* A suite of ten Unix programs written in C is also among the small-sized programs studied. Since these programs have been so thoroughly used, they can serve as a reliable basis to evaluate the behavior of fault-injected programs derived from them. Table IX provides a summary of this suite.

[5]"Performs better" implies "examining less code to locate the first statement, which contains the bug," "performs the same" implies "examining the same amount of code (but not necessarily the same code)," and "performs worse" implies "examining more code."

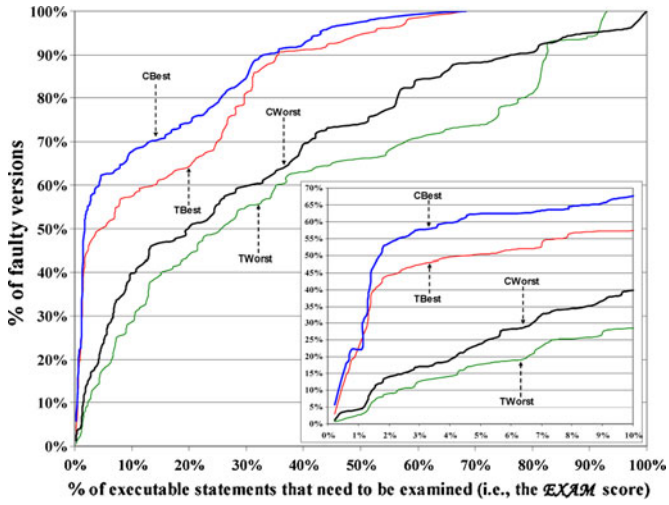


Fig. 2. Effectiveness comparison on the Unix suite.

TABLE X
PAIRWISE COMPARISON BETWEEN CBT AND TARANTULA FOR THE UNIX SUITE

| | CBest versus TBest | CWorst versus TWorst | CWorst versus TBest |
|---|---|---|---|
| More effective | 40.12%(69) | 62.21%(107) | 17.44%(30) |
| Same effectiveness | 59.30%(102) | 26.74%(46) | 8.72%(15) |
| Less effective | 0.58%(1) | 11.05%(19) | 73.84%(127) |

More descriptions of the test case generation, fault set, and erroneous program preparation can be found in [28].

Fig. 2 gives the effectiveness of Tarantula and CBT for the Unix suite, where all the legends, the horizontal axis, and the vertical axis have the same meaning, as in Fig. 1.

As with the Siemens suite, the curves for CBest and TBest indicate that CBT performs better than Tarantula in the best case. The same observation can also be made when comparing the two worst cases (CWorst and TWorst). Table X in a manner similar to Table VII presents a pairwise comparison between CBT and Tarantula on the Unix suite. We observe that CBest is only worse than TBest for 1 out of a total of 172 faulty versions. For other faulty versions, CBest is always better than, or at least equally as effective as TBest. In the case of the comparison between CWorst and TWorst, we find that for the majority of the faulty versions (62.21%), CWorst performs better than TWorst. Moreover, CWorst is also better than TBest on 30 faulty versions.

Similar to Table VIII, the total number of statements that must be examined in order to locate all of the bugs in the Unix suite is presented in Table XI. We observe that CBest is significantly less than TBest and that CWorst is also significantly less than TWorst, meaning that less effort is to be spent on locating the bugs using CBT (as opposed to Tarantula) in either case. Once again there may not be any subset relationship between the statements examined by CBT and those by Tarantula.

Based on the data provided we conclude that CBT performs better than Tarantula on the Unix suite.

TABLE XI
TOTAL STATEMENTS EXAMINED FOR ALL FAULTY VERSIONS OF THE UNIX SUITE

| CBest | CWorst | TBest | TWorst |
|---|---|---|---|
| 2462 | 6030 | 3364 | 7629 |

### C. Programs of a Relatively Larger Size

The programs in the Siemens and the Unix suites are small-sized programs in the sense that they are all less than 1000 lines of code. To further evaluate the effectiveness of CBT, we now apply it to larger programs (in terms of the LOC), namely—the *space program*, *grep*, *gzip*, and *make*.

The space program developed at the European Space Agency provides an interface that allows the user to describe the configuration of an array of antennas [4]. The correct version, the 38 faulty versions, and a suite of 13585 test cases used in this study are downloaded from a software-artifact infrastructure repository [10].

The source code of version 2.2 of the grep program is downloaded from [10], along with 18 faulty versions and a suite of 470 test cases. Compared with the study in [18], where none of these bugs could be detected by any test case in the suite, we are able to detect four bugs in our environment. Two additional bugs injected by Liu *et al.* [18] are also used. Liu *et al.* [18] argue that although faults are manually injected, they do resemble realistic logic errors. An additional 13 bugs are injected by us following the same approach, bringing the total number of faulty versions to 19. See Section VIII-E for more discussions on mutation-based fault injection.

Version 1.1.2 of the gzip program with 16 seeded bugs and 217 test cases are downloaded from [10]. Nine faults are excluded from the study, since none of the test cases fails on them and six test cases are discarded because they cannot be executed in our environment. As stated earlier, we follow a similar approach as described in [18] to inject 21 bugs in addition to the 7 (16 minus 9) usable original bugs. Thus, in total, 28 faulty versions are used. Version 3.76.1 of make is downloaded from [10] along with 793 test cases and 19 faulty versions of the program. Of these, 15 faulty versions are excluded as they contain bugs, which cannot be detected by any of the downloaded test cases in our environment. Using the aforementioned fault-injection approach, we generate an additional 27 bugs for a total of 31 faulty versions to be used.

A summary of these four programs is presented in Table XII. A list of all additional bugs that have been seeded for the purposes of the experiments are available upon request. As observed from Table XII, we find that these programs are relatively larger than those of the Siemens and Unix suites, and all of them have an LOC count (and executable statement count) that numbers in the thousands. This also allows us to observe the effectiveness of CBT on larger programs.

Figs. 3–6 illustrate the effectiveness of CBT when compared to Tarantula across the space, grep, gzip, and make programs, respectively. Upon observation of the curves, we find that not only is CBest better than TBest and CWorst better than TWorst,

TABLE XII
SUMMARY OF THE SPACE, GREP, GZIP, AND MAKE PROGRAMS

| Program | Number of faulty versions | LOC | Number of executable statements | Number of test cases |
|---|---|---|---|---|
| space | 38[a] | 9126 | 3657 | 13585 |
| grep | 19 | 12653 | 3306 | 470 |
| gzip | 28 | 6573 | 1670 | 211 |
| make | 31 | 20014 | 5318 | 793 |

[a]Three of the 38 faulty versions not used.

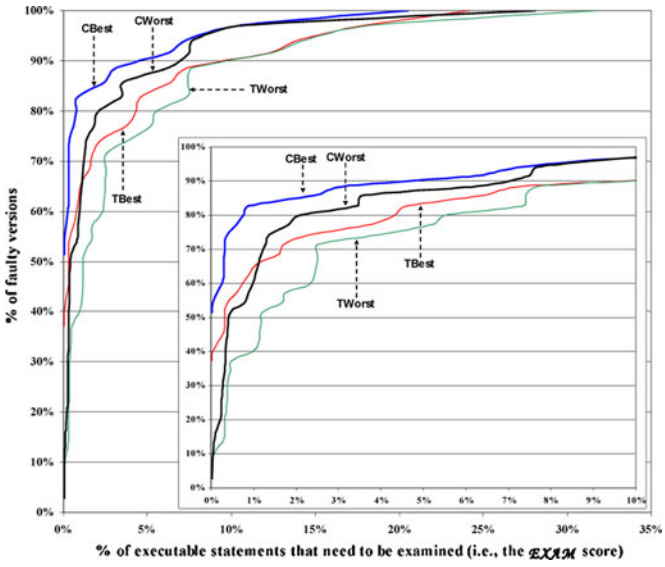


Fig. 3. Effectiveness comparison on the space program.

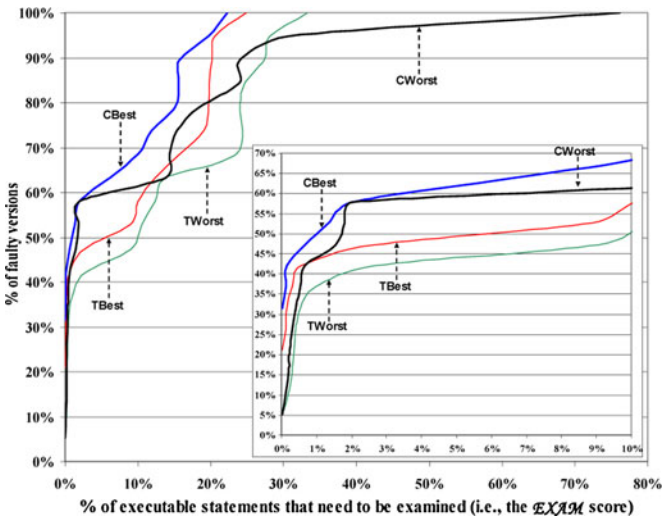


Fig. 4. Effectiveness comparison on the grep program.

but on several occasions CWorst is also better than TBest (which is especially true in the case of the space program).

Table XIII presents the pairwise comparison between CBT and Tarantula, across the space, grep, gzip, and make programs. We observe that when comparing CBest and TBest, there is only one case, where CBest is less effective than TBest (on one faulty version of space) and for all other faulty versions, CBest is always more effective than or as effective as TBest. The same is true when comparing CWorst and TWorst because only for

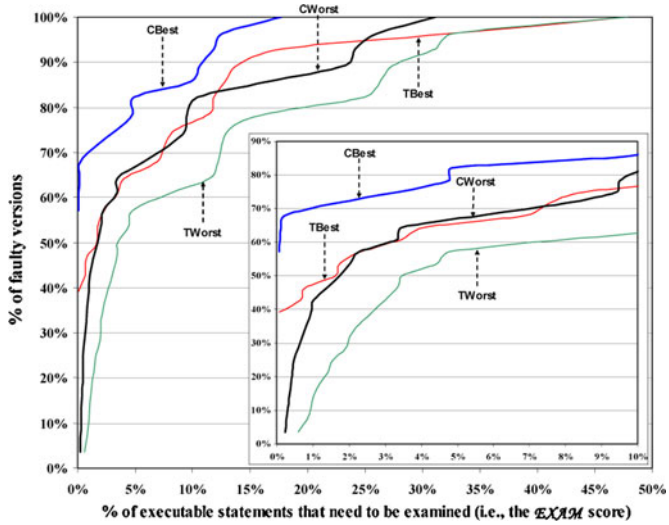


Fig. 5. Effectiveness comparison on the gzip program.

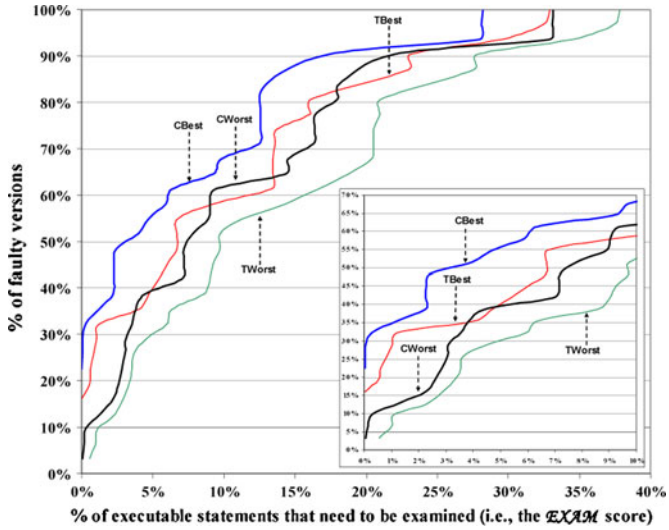


Fig. 6. Effectiveness comparison on the make program.

one faulty version of grep is CWorst ever worse than TWorst. For all other faulty versions, CWorst is always better than or at least as effective as TWorst. We also observe as with the smaller sized programs that there are several faulty versions for which CWorst even performs better than TBest.

The comparison between CBT and Tarantula in terms of the total number of statements that need to be examined in order to locate all of the faults is presented in Table XIV. Based on the data, we observe that not only is CBest better than TBest and CWorst better than TWorst for every program, but in the case of the space and gzip programs, CWorst is actually better than TBest. This means that even in the worst case, CBT can lead programmers to all of the bugs by examining a fewer number of statements than Tarantula can even in the best case. In every case, there may not be a subset relationship between the statements examined by CBT and those by Tarantula.

Based on the comparisons between the effectiveness of CBT and Tarantula across both small- and large-sized programs, we

TABLE XIII
PAIRWISE COMPARISON ON THE SPACE, GREP, GZIP, AND MAKE PROGRAMS

| Program | | CBest versus TBest | CWorst versus TWorst | CWorst versus TBest |
|---|---|---|---|---|
| space | More effective | 51.43%(18) | 62.86%(22) | 37.14%(13) |
| | Same effectiveness | 45.71%(16) | 37.14%(13) | 2.86%(1) |
| | Less effective | 2.86%(1) | 0 | 60.00%(21) |
| grep | More effective | 63.16%(12) | 68.42%(13) | 42.11%(8) |
| | Same effectiveness | 36.84%(7) | 26.32%(5) | 5.26%(1) |
| | Less effective | 0 | 5.26%(1) | 52.63%(10) |
| gzip | More effective | 50.00%(14) | 67.86%(19) | 21.43%(6) |
| | Same effectiveness | 50.00%(14) | 32.14%(9) | 3.57%(1) |
| | Less effective | 0 | 0 | 75.00%(21) |
| make | More effective | 74.19%(23) | 90.32%(28) | 29.03%(9) |
| | Same effectiveness | 25.81%(8) | 9.68%(3) | 0 |
| | Less effective | 0 | 0 | 70.97%(22) |

TABLE XIV
TOTAL NUMBER OF STATEMENTS EXAMINED FOR THE SPACE, GREP, GZIP, AND MAKE PROGRAMS

| Programs | CBest | CWorst | TBest | TWorst |
|---|---|---|---|---|
| space | 2016 | 3093 | 3876 | 5094 |
| grep | 4005 | 7443 | 5793 | 7812 |
| gzip | 1314 | 3091 | 3110 | 5032 |
| make | 12403 | 18142 | 16890 | 23468 |

TABLE XV
RELATIVE IMPROVEMENT OF CBT OVER TARANTULA

| Programs | CBest | TBest | CBest/TBest | CWorst | TWorst | CWorst/TWorst |
|---|---|---|---|---|---|---|
| Siemens suite | 2005 | 2453 | 81.74% | 2873 | 3311 | 86.77% |
| Unix suite | 2462 | 3364 | 73.19% | 6030 | 7629 | 79.04% |
| space | 2016 | 3876 | 52.01% | 3093 | 5094 | 60.72% |
| grep | 4005 | 5793 | 69.14% | 7443 | 7812 | 95.28% |
| gzip | 1314 | 3110 | 42.25% | 3091 | 5032 | 61.43% |
| make | 12403 | 16890 | 73.43% | 18142 | 23468 | 77.31% |
| $\sum$ | 24205 | 35486 | **68.21%** | 40672 | 52346 | **77.70%** |

TABLE XVI
RESULTS OF THE WILCOXON SIGNED-RANK TEST

| Programs | $H_A$: TBest requires the examination of more statements than CBest | $H_A$: TWorst requires the examination of more statements than CWorst |
|---|---|---|
| Siemens suite | $2.473 \times 10^{-11}$ | $9.522 \times 10^{-10}$ |
| Unix suite | $3.726 \times 10^{-17}$ | $2.244 \times 10^{-13}$ |
| space | $5.722 \times 10^{-6}$ | $2.384 \times 10^{-7}$ |
| grep | $2.441 \times 10^{-4}$ | 0.00671 |
| gzip | $6.104 \times 10^{-5}$ | $1.907 \times 10^{-6}$ |
| make | $1.192 \times 10^{-7}$ | $3.725 \times 10^{-9}$ |
| All Programs | $1.781 \times 10^{-31}$ | $2.356 \times 10^{-34}$ |

can conclude that CBT is more effective than Tarantula in terms of its fault-localization capabilities.

### *D. Relative Improvement Over Tarantula*

While the comparisons presented in Sections IV-A and B clearly indicate that CBT is more effective than Tarantula for fault localization, another important question that needs to be answered is—"Is the difference significant?" To address this, we present a comparison between CBT and Tarantula that quantifies the percentage improvement CBT has over Tarantula. A suitable basis for such a comparison is the total number of statements that each technique must examine in order to locate bugs for all the faulty versions of each program, as well as all the faulty versions of all programs. Such a comparison between CBT and Tarantula is presented across all of the programs in Table XV.

Based on the data in Table XV, we find that in the best case, consider all of the programs, the number of statements that need to be examined by CBT (24 205) to locate all the bugs is 68.21% of the number of statements that have to be examined by Tarantula (35 486). In the worst case, the percentage is 77.70% (=40672/52346). It is important to point out here that when we say CBest only requires the examination of 68.21% of the number of statements that TBest requires the examination of, this does not imply that the statements examined by CBest would necessarily be a subset of those examined by TBest. This is also true in the case of CWorst and TWorst. Any comparisons drawn here are only on the size of the sets of statements that need to be examined by either technique, and not on the sets themselves.

Additionally, in order to better evaluate the performance of CBT with that of Tarantula using sound statistics, we make use of the Wilcoxon signed-rank test. The Wilcoxon signed-rank test makes use of the sign and the magnitude of the rank of the differences between pairs of measurements, where the formal null hypothesis is that the population distribution of differences is symmetrical about some value $D_0$ (where $D_0$ is usually set to 0) [22]. Translated into our context, essentially the null hypothesis would be that the distribution of the differences between the total number of statements that need to be examined, to locate fault(s), by CBT and Tarantula is symmetric around 0, i.e., CBT and Tarantula are equally effective. We evaluate the one-tailed alternate hypothesis that Tarantula requires the examination of more statements than CBT, i.e., CBT is more effective than Tarantula.

Table XVI presents the results of the Wilcoxon signed-rank test in terms of the resultant $p$-values (the entry in each cell) and splits comparisons into those between CBest and TBest and between CWorst and TWorst.

We find that all $p$-values are considerably small for each of the tests, implying that the alternative hypotheses can be generally accepted, which implies that CBT requires the examination of fewer statements to locate faults than Tarantula.

## V. Comparison With Two Other Statistical Fault-Localization Techniques

Liblit *et al*. [17] presented a technique for statistical debugging (Liblit05) as did Liu *et al*. [18] (SOBER). Since our proposed CBT is also categorically a statistical technique for fault localization, we now compare these three techniques both qualitatively and quantitatively.

### A. From a Qualitative Perspective

One of the biggest differences between the technique proposed in this paper, and the SOBER and Liblit05 is that we rank suspicious statements, whereas the other two rank suspicious predicates that can then be examined in order of suspiciousness to locate the faults. Yet another difference between these techniques and CBT is in terms of the code-examination strategy that the techniques employ. In the case of SOBER, if the bug is not in the initial set of predicates to be examined, additional predicates need to be included via a breadth-first search on the program dependence graph of the program being debugged. In contrast, such a search is not required by CBT as all statements of the program are ranked only once based on their suspiciousness. Thus, in the case of CBT, code is examined along the final ranking that is produced and not along dependencies as is done in the case of SOBER and Liblit05. In fact CBT does not even require any static dependency information between the program elements in order to form its final ranking.

From a more statistical point of view, SOBER uses a hypothesis test to indirectly quantify the difference between $f_P(X|\text{Correct})$—the evaluation bias observed from the correct executions and $f_P(X|\text{Incorrect})$—the evaluation bias observed from the incorrect executions, where $X$ is the random variable representing the evaluation bias of predicate $P$. Instead CBT uses a hypothesis test to provide a *reference* of "dependence/independence" between the execution results and the coverage of each statement. Additionally, the test statistic employed in SOBER asymptotically conforms to a normal distribution under the null hypothesis, and a transformation of the probability density function of the normal distribution is used as the similarity function to compute the fault-relevance ranking score for each predicate $P$. On the other hand, our test statistic $\chi^2$ has an approximately chi-square distribution, and we use a transformation of this statistic [see (2)] to measure the degree of association for subsequent analysis to compute statement suspiciousness.

These stark differences between CBT and techniques, such as SOBER and Liblit05, also serve to highlight the novelty of our proposed approach, though we strongly emphasize that the novelty of CBT is not just with respect to the aforementioned techniques. For example, in no way was the SOBER technique modified to produce the CBT technique and there is no link between the two in terms of lineage. Any similarities that are shared are purely coincidental or due to commonalities in principles and intuition.

### B. From a Quantitative Perspective

We also present a quantitative comparison among CBT, SOBER, and Liblit05 based on data reported in [18] on the Siemens suite. It is important to note that both Liblit05 and SOBER are sensitive to the number of predicates (the parameter $k$) chosen to form the fault-localization report. In [18], the results provided are based on the choice of $k = 5$, which *empirically* evaluates to the best case. As stated by the authors, "too few predicates (e.g., $k = 1$) may not convey enough information for fault localization, while too many predicates (e.g., $k = 9$) are in themselves a burden for developers to examine and, thus neither of them leads to the best result." This is one of the weaknesses of such fault localizers, as there is no explanation on how to estimate the ideal value of $k$. It is only *empirically* with the variation of the value of $k$ that a good value of this parameter can be established (relative to the bad values). In [18], Liu *et al*. performed a quality comparison of the effectiveness of Liblit05 and SOBER across $k$ values ranging from 1 to 9, and only via such a comparison, it is established that $k = 5$ evaluates to the best case (among the $k$-values evaluated).

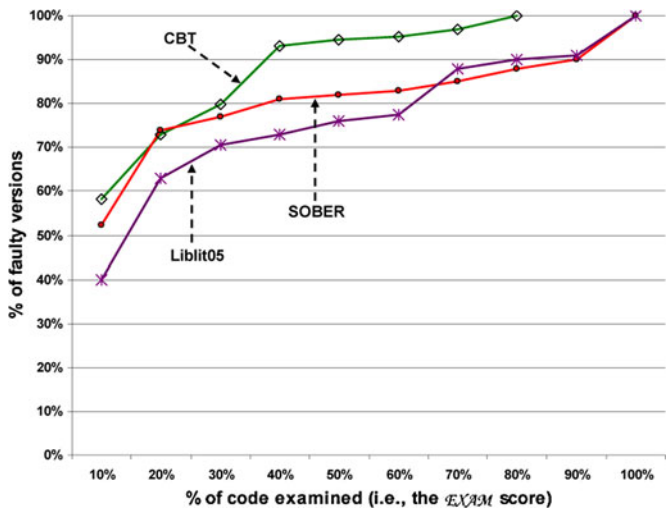


Fig. 7. Effectiveness of CBT versus SOBER and Liblit05 on the Siemens suite.

Based on Fig. 1 provided in Section IV-B1 on the Siemens suite, we find that CBT can identify 75 of 129 faults (i.e., 58.14% of the faults) by examining no more than 10% of the code in the best case. In contrast, SOBER in the best case identifies 52.31% of the faults, and Liblit05 in the best case identifies 40% of the faults by examining no more than 10% of the code. Additionally, we are able to retrieve data from the curves in Fig. 4(b) of [18] regarding the best case effectiveness of SOBER and Liblit05. Since the exact numbers are not reported in [18], we derive them directly from the curves. To minimize human error, two Ph.D. students retrieved the data independently, and their data was cross validated for accuracy against the curves. Fig. 7 presents the comparison among CBT, SOBER, and Liblit05 on the Siemens suite.

Based on the curves, we observe that on the Siemens suite, CBT is always better than Liblit05. In addition, except for when 20% of the code is examined and SOBER locates 73.85% of the faults and CBT 72.87% of the faults, CBT performs better than SOBER in terms of the fault-localization effectiveness (i.e., locating program bugs by examining less code).

Such a comparison between CBT and SOBER is fair as in each case, the best-case effectiveness of each technique has been considered. A similar comparison has not been presented for the "worst-case effectiveness" for two reasons: First, there is no value of $k$ that has been established as the worst case for SOBER and Liblit05. The value, which truly conforms to the worst case may not have even been considered by the authors in [18]. Second, while the authors do present figures corresponding to the $k$-values 1 through 9 in [18], they restrict themselves to only provide information on the percentage of faults located, while examining a maximum of 20% of the code [18, Fig. 5]. Such a restriction by the authors makes it impossible for us to fairly compare the worst case effectiveness of each technique, as data on the amount of code that needs to be examined to locate all the faults is not provided. To understand why this data is required for a fair comparison, consider the case, where a fault localizer A can locate 50% of the faults by examining only 20% of the code. Let us also consider a fault localizer B that can only locate 10% of the faults by examining 20% of the code. Seemingly, A is more effective than B. However, if we increase the amount of code examined to 25%, we find that B can locate 80% of the faults, whereas A can only locate 55% of the faults. Furthermore, B is able to locate all of the faults by examining 30% of the code, whereas A requires the examination of 90% of the code to locate all of the faults. Thus, we had limited ourselves to examining only 20% of the code, the comparison between the fault localizers would clearly have been unfair.

We conclude that CBT does not just result in higher fault-detection effectiveness than techniques, such as Tarantula, but in other statistical fault-localization techniques, such as Liblit05 and SOBER, as well.

## VI. Evaluation Across Different Perspectives

The $\mathcal{EXAM}$ score is a useful way to compare two or more fault-localization techniques with each other. However, there may be other considerations that need to be kept in mind when designing a fault localizer. In this section, we identify a few such considerations, and evaluate CBT and Tarantula with respect to each of them.

### A. Applicability to Object-Oriented Languages

Theoretically, since CBT utilizes only coverage information and test case execution results, it is indifferent toward the programming language of the program being debugged. However, since our subject programs thus far have all categorically been C programs, we decided to apply CBT to a Java program to verify the quality of the crosstab-based fault localization on object-oriented programs as well.

Ant is a Java-based build tool supplied by the open source Apache project, available at http://ant.apache.org. It is similar to the tool *make* but uses a model wherein extensions are implemented as Java classes rather than shell-based commands. Configuration files are XML-based, calling out a target tree, where various tasks get executed. Each task is run by an object that implements a particular task interface. The source code for version 1.6 beta of Ant as well as a set of six single-bug versions

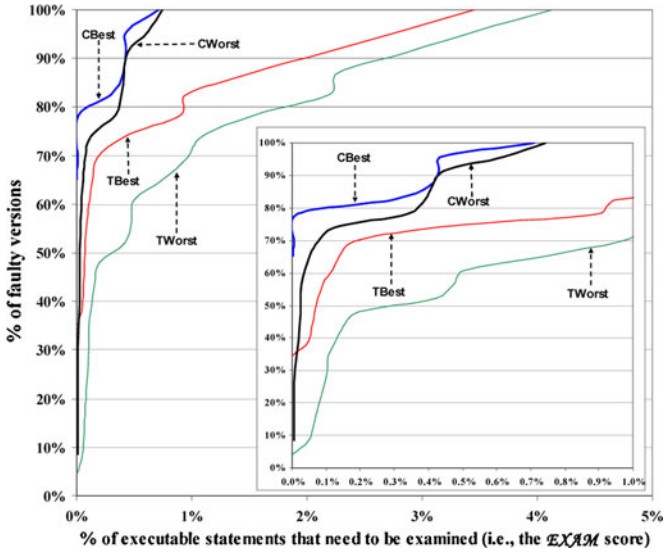


Fig. 8. Effectiveness comparison on the Ant program.

TABLE XVII
Pairwise Comparison on the Ant Program

| | CBest versus TBest | CWorst versus TWorst | CWorst versus TBest |
|---|---|---|---|
| More effective | 15 | 22 | 14 |
| Same effectiveness | 8 | 1 | 1 |
| Less effective | 0 | 0 | 8 |

and 877 test cases are downloaded from [10]. After merging multiline statements, etc., we record 45 701 lines of code spread across different classes for the entire body of source. Clover [11] is used to instrument the code and collect coverage information. Of the 877 test cases, four of them cannot be executed in our environment and two of them provide different outputs on instrumented (by Clover) and noninstrumented versions of the same program. Hence, these six test cases are discarded, and 871 test cases are used in the experiment. In addition to the six downloaded single-bug versions, as with the make, grep, and gzip programs, we seed 17 other faults, which give us a total of 23 faulty versions to work with. The test case executions are run on a PC with a 2.13 GHz Intel Core 2 Duo CPU and 8 GB physical memory, and the Java compiler version is 1.5.0_06.

Fig. 8 presents a comparison between CBT and Tarantula for the Ant program based on the $\mathcal{EXAM}$ score (the zoom-in is provided at the 1% level instead of the 10% level). We observe that not only is CBest better than TBest and CWorst better than TWorst, but CWorst is also better than TBest.

Tables XVII and XVIII provide the pairwise comparison and the comparison in terms of the total number of statements required to locate all the faults between CBT and Tarantula, respectively, on the Ant program. From Table XVII, we observe that CBest is always more effective than, or at least as effective as, TBest for all of the faulty versions, and the same is true of the comparison between CWorst and TWorst. Additionally, we observe that CWorst is also better than TBest for the majority of the faulty versions. Based on Table XVIII, we observe that CBT

TABLE XVIII
TOTAL NUMBER OF STATEMENTS EXAMINED FOR ALL THE 23 FAULTY VERSIONS OF THE ANT PROGRAM

| CBest | CWorst | TBest | TWorst |
|---|---|---|---|
| 1076 | 1578 | 5964 | 9935 |

can lead the programmer to all of the faults by examining a fewer number of statements than Tarantula, regardless of whether the best or worse case is considered. In fact, even CWorst is better than TBest, requiring the examination of only about 26.46% (1578/5964) of the statements (that TBest does) to locate all the faults. All the comparisons are based on the size of the sets of statements that need to be examined instead of the sets themselves. There may not be any subset relationship between the statements examined by CBT and those by Tarantula.

Based on the three modes of comparison (see Fig. 8, and Tables XVII and XVIII), we conclude that CBT still performs better than Tarantula even in the case of an object-oriented programming language-based program, such as Ant. In fact, based on the Ant data, we observe that not just is CBT better than Tarantula when the best case is compared with the best, and the worst with the worst; rather, often, the worst case of CBT is still significantly better than the best case of Tarantula.

### B. Timing Considerations

In addition to being effective by examining less code to find a bug, it is important that each fault-localization technique also be efficient. The time required to use a fault-localization technique to identify suspicious code includes two parts. Part I involves data collection. Part II uses the technique with the data collected in Part I to actually prioritize which portion of the program should be examined to locate the bugs. Both CBT and Tarantula operate on the same set of data, namely, whether the program execution fails or succeeds with respect to each test case, and whether each statement is covered by this test case. The coverage information is saved in an execution trace created by a revised version of $\chi$Suds [35] as discussed earlier and the success/failure information is saved in a separate text file. Part II can be further subdivided into two parts. Part IIA is the time taken to read in the coverage information and the execution results collected in Part I from files into memory and save them in the same format as Table III. Part IIB is the computational time to retrieve the necessary information from memory to compute the intermediate values required by CBT and Tarantula, and the suspiciousness of each executable statement. The time required for Parts I and IIA is the same for both techniques. Hence, these two parts are not included in the following comparison.

Table XIX gives the average computational time taken by CBT and Tarantula with respect to all the faulty versions of each program. The first step to obtain such data is to perform the computation 100 times for each faulty version of a given program. This way, any bias can be significantly reduced if not eliminated entirely. The second step is to add all the computational times together, divide the sum by 100, and then divide by the number of faulty versions of that program. For example, the average computational time of the ten faulty versions of "schedule2" is 0.002984 s for Tarantula and 0.002950 s for CBT.

TABLE XIX
TIME-BASED PERFORMANCE ANALYSIS OF CBT AND TARANTULA

| Program | | Avg. computational time for a program in seconds[a] | |
|---|---|---|---|
| | | Tarantula | CBT |
| Siemens Suite | print_tokens | 0.006794 | 0.006826 |
| | print_tokens2 | 0.006874 | 0.006957 |
| | replace | 0.011004 | 0.011719 |
| | schedule | 0.003380 | 0.003453 |
| | schedule2 | 0.002984 | 0.002950 |
| | tcas | 0.001395 | 0.001472 |
| | tot_info | 0.001622 | 0.001710 |
| Unix suite | Cal | 0.000309 | 0.000401 |
| | Checkeq | 0.000334 | 0.000411 |
| | Col | 0.000562 | 0.000632 |
| | Comm | 0.000764 | 0.000886 |
| | Crypt | 0.000380 | 0.000471 |
| | Look | 0.000478 | 0.000377 |
| | Sort | 0.005394 | 0.005439 |
| | Spline | 0.001239 | 0.001195 |
| | Tr | 0.000969 | 0.000998 |
| | Uniq | 0.000614 | 0.000755 |
| space | | 2.445176 | 2.329287 |
| grep | | 0.047177 | 0.046493 |
| gzip | | 0.010355 | 0.011800 |
| make | | 0.239727 | 0.224827 |
| Ant | | 0.357059 | 0.362383 |

[a]Computational time may be further reduced because the scripts used for computing the suspiciousness are written in Java and not optimized.

Based on the data provided in Table XIX, we observe that the difference is almost always negligible. Sometimes CBT is faster than Tarantula, and sometimes it is the other way around. The maximum difference ever recorded is for the space program when the computational time for CBT is 0.116 s less than that of Tarantula. Thus, we observe that even though CBT is more effective than Tarantula at fault localization, it does so without sacrificing efficiency.

### C. Sensitivity to Test Suite Size

It is impractical to assume that large comprehensive test sets are always available, and therefore, it is important to investigate how sensitive a fault-localization technique is to test suite size. Such an experiment is performed in [18] using the SOBER technique on the Siemens suite, where *inadequate*[6] test suites are simulated by sampling (without replacement) from the original test suites. We also conduct such an experiment in a similar manner on the programs of the Siemens suite, comparing the effectiveness of CBT and that of Tarantula. Given a sampling rate $\beta$ ($0 < \beta \leq 1$) and an original test suite size of $N$, we randomly select $N \times \beta$ test cases without replacement to produce a test suite $T_\beta$ (we refer to such a test suite as a $\beta$-sampled test suite). If a $T_\beta$ does not contain at least one failed test case it is discarded, and the process is repeated until a $T_\beta$ containing at

[6]Liu *et al.* [18] did not define the criterion with respect to which a test suite may be adequate or inadequate. Being a controlled study, it is assumed that the test suites downloaded for experiment represent "adequate" test suites. It is also understood that the sampled test suite becomes more "inadequate" as the sampling rate becomes smaller.

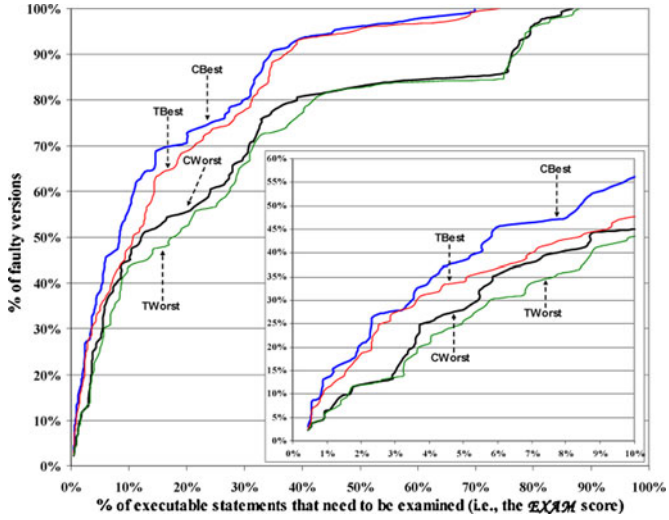


Fig. 9. Effectiveness comparison on the Siemens suite when 10% of the test suite is sampled.

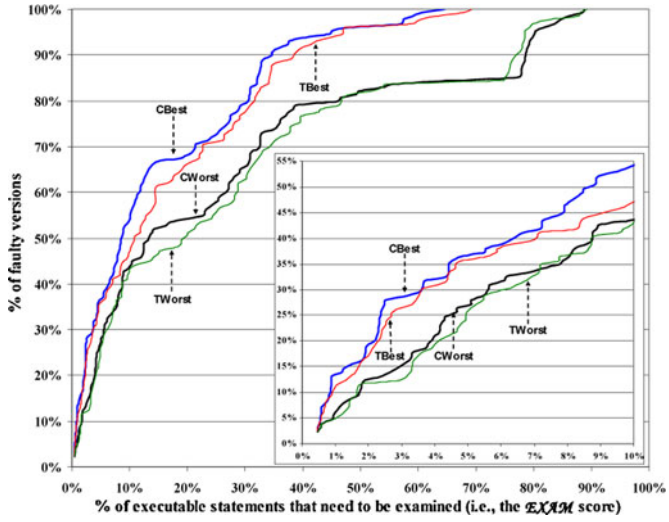


Fig. 10 Effectiveness comparison on the Siemens suite when 5% of the test suite is sampled.

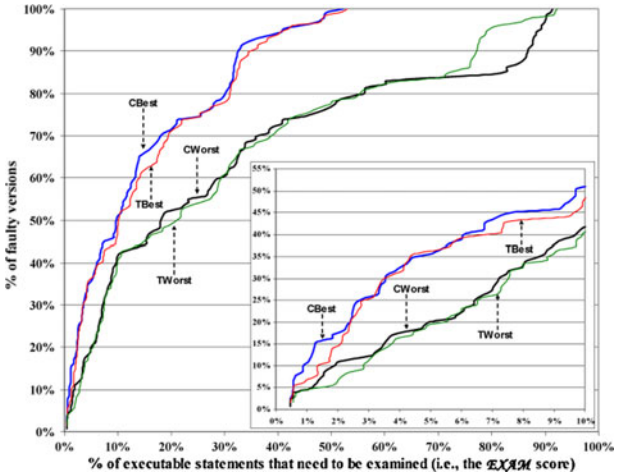


Fig. 11. Effectiveness comparison on the Siemens suite when 1% of the test suite is sampled.

TABLE XX
TOTAL STATEMENTS EXAMINED TO LOCATE ALL FAULTS OF THE SIEMENS SUITE

| Sampling Rate (β) | CBest | CWorst | TBest | TWorst |
|---|---|---|---|---|
| 10% | 2073.2 | 3127.3 | 2437 | 3470.75 |
| 5% | 2144.5 | 3329.65 | 2470.45 | 3600.9 |
| 1% | 2096.05 | 3889.4 | 2258.5 | 3926.55 |

least one failed test case is produced. Then, both Tarantula and CBT are evaluated using only the information in $T_\beta$.

We note that one $\beta$-sampled test suite may differ considerably from another, and therefore, the fault-localization results obtained using a test suite of a certain size may vary from the results obtained from another test suite of the same size. In order to reduce such bias, evaluations across any sampling rate are conducted 100 times, and the results averaged. We choose sampling rates of 10%, 5%, and 1%, respectively. Note that $T_{100\%}$ corresponds to using the original test suite in its entirety, and therefore, the data for $T_{100\%}$ is the same as what is presented in Section IV-B1.

Figs. 9–11 show the comparison (by means of the $\mathcal{EXAM}$ score) between CBT and Tarantula on the Siemens suite when 10%, 5%, and 1% of the test suite is sampled, respectively. From the curves, we observe that in general CBT performs better than Tarantula, and this becomes easier to see when we consider the zoom-ins.

In terms of the total number of statements that need to be examined to find all of the faults, this comparison is presented in Table XX. Note that there may be fractional values in the table, whereas the number of statements that are examined should indeed be values of integer form. The fractional component arises because, as pointed out earlier in this section, for each sampling rate $\beta$, we generated 100 distinct $\beta$-sampled test suites and averaged the results. From the table, we observe that irrespective of the sampling rate and irrespective of whether the best or worst case is considered, CBT always performs better than Tarantula in that a fewer number of statements need to be examined by CBT compared with Tarantula in order to locate all of the faults. We reiterate that there may not be any subset/superset relationship between the statements examined by CBT and those by Tarantula.

We also make other interesting observations based on the data. Intuitively, the less information a fault-localization technique has available to work with (in our case, coverage and execution result for each selected test case), the poorer the performance of the fault-localization technique is expected to be. The curves presented in Figs. 9–11, as well as the data in Table XX somewhat ratify this intuition. However, we notice that for either technique, the worst case effectiveness seems to decrease with a decreasing sampling rate (i.e., more statements need to be examined to locate the faults if the sampling rate is reduced), while this is not true for the best case effectiveness. Indeed from Table XX, we find that CBest when 1% of the test suite is sampled is better than CBest when 5% of the test suite is sampled. The same is true for Tarantula because TBest at a 1%

sampling rate is better than TBest at a 5% sampling rate. This can introduce a misleading conclusion that a smaller sampling rate is better than a larger sampling rate.

We conjecture that the reason for such observations is that because of having less information to work with, fault-localization techniques are not able to distinguish statements from one another well with low sampling rates. The same suspiciousness value gets assigned to more statements [including faulty statement(s)] than normal, resulting in more ties. More ties mean that the best effectiveness may go up, while the worst effectiveness may go down. To better illustrate this, consider the trivial situation, where we assign every statement the same suspiciousness, thereby tying every statement in the fault-localization ranking. Consider that our program has $m$ statements, and therefore, in the best case, we will only examine one statement to find the fault, and in the worst case, we need to examine all of the $m$ statements. Thus, we have a very high best case effectiveness and a very low worst. The ability of a fault-localization technique to distinguish one statement from another in terms of its likelihood of being faulty is crucial, and the effect to reduce the sampling rate (i.e., reducing the size of the test suite) is to hamper the ability of techniques to distinguish statements from one another.

Finally, we note that even though the amount of code that needs to be examined using CBT with a lower sampling rate (see Table XX) is more than that which would need to be examined by CBT with the entire test suite (see Table VIII), the effectiveness of CBT does not really suffer all that much compared with the reduction in sampling rate. This allows us to have confidence in the robustness of CBT and its ability to perform well even in harsh situations, where large-sized test suites may not be available.

## VII. Programs With Multiple Bugs

Thus far, the discussion has been on programs that only contain a single bug per faulty version and such an assumption has also been made by several other fault-localization studies (such as [5], [14], [18], and [24]). However, CBT can also be easily applied to effectively locate faults for programs with multiple bugs as well. In this section, we discuss how this might be done and also present an illustrative case study.

When working with programs with multiple bugs, a typical strategy is to locate and fix a single fault, and then rerun test cases to detect subsequent failures, whereupon the next fault is located and fixed. The process may continue until test case failures are no longer observed, and we can deduce that there are no more faults present in the program that can be detected by the current test suite. This approach is followed in studies, such as [36], where Yu *et al*. evaluate the effects of test suite reduction on fault localization, and for programs with multiple faults, they define their evaluation metric for fault localization (which they term as "expense") as the percentage of code that must be examined (via a ranking) to locate the first fault,[7] as they argue that this is the first fault developers would fix.

[7]Even though it has been defined in a multibug setting, we note that the "expense" score in [36] is very similar to the $\mathcal{EXAM}$ score used here.

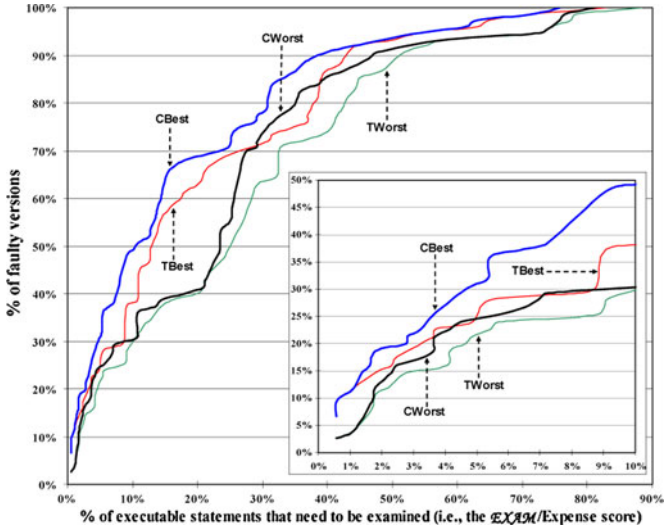


Fig. 12. Effectiveness comparison between CBT and Tarantula for the multibug versions of the Siemens suite.

Thus, CBT can also be applied to programs with multiple bugs in such a manner, and in accordance, we conduct a comparison between CBT and Tarantula using this strategy. As per [36], both CBT and Tarantula are evaluated on the basis of the "expense" required to find only the first fault in the ranking. Note that since the rankings based on CBT may be quite different from those of Tarantula, it is not necessary that the first fault located by CBT be the same as the first fault located by Tarantula. Multibug versions of the subject programs are created via a combination of several of the single-bug versions that are available to us. For example, a two-bug version of a program can be created by seeding each of the faults of two single-bug 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-bug versions available (see Section IV-B1), which can be combined in a variety of ways to produce many different multibug versions. A total of 75 multibug versions are created based on combinations of the single-bug programs, and these range from faulty versions with two bugs in them to those with five bugs. Fig. 12 presents the comparison between CBT and Tarantula using the strategy described in this section on the multibug versions of the Siemens suite.

Based on the figure, we observe that CBest is better than TBest and CWorst is better than TWorst, and this is especially evident in the case of the zoom-ins. In fact, CWorst is often comparable to TBest. This clearly demonstrates the superiority of CBT over Tarantula when using the $\mathcal{EXAM}$ (or "expense") score as a means of comparison.

We also present the total number of statements that need to be examined by CBT and Tarantula using this strategy in Table XXI. Note that these numbers do not represent the total number of statements that require examination (using either technique) to locate all of the faults in all of the 75 multibug programs, but rather the number of statements that must be examined to find the first fault in all of the multibug programs. Using the data in Table XXI as the means of comparison, we

TABLE XXI
TOTAL NUMBER OF STATEMENTS EXAMINED FOR THE MULTIBUG VERSIONS OF THE SIEMENS SUITE BY CBT AND TARANTULA

| CBest | CWorst | TBest | TWorst |
|---|---|---|---|
| 1413 | 1839 | 1654 | 2067 |

TABLE XXII
NUMBER OF STATEMENTS THAT NEED TO BE EXAMINED USING TARANTULA AND CBT FOLLOWING THE ONE-BUG-AT-A-TIME STRATEGY

| Iteration | Number of statements examined | | | |
|---|---|---|---|---|
| | CBest | CWorst | TBest | TWorst |
| 1 | 1 | 3 | 2 | 4 |
| 2 | 2 | 5 | 33 | 36 |
| 3 | 61 | 61 | 87 | 87 |
| 4 | 99 | 108 | 104 | 113 |
| 5 | 2 | 2 | 5 | 5 |
| Total | 165 | 179 | 231 | 245 |

again observe that CBT is more effective than Tarantula in that fewer statements need to be examined to find the first fault in all multibug programs (irrespective of whether the best case or the worst cases is considered).

It is important to point out that while the "expense" score defined in [36] refers to the percentage of code that needs to be examined in order to find only the first bug in a multibug program (as this is the first fault that would be fixed), it is really part of a bigger process that involves sequentially locating and fixing all bugs (whose effects are observable) that may reside in the subject program (as per the description provided at the beginning of this section). This strategy is referred to as the *one-bug-at-a-time* strategy, and the current expense score assesses the fault-localization effectiveness with respect to just the first step (or iteration) of the strategy. Since each iteration of the process involves locating a particular fault, there is an "expense" score associated with every iteration, and the number of iterations would equal the number of faults. Essentially, the cumulative number of statements examined to find a total of $k$ faults can be quantified as $\sum_{i=1}^{k} |\alpha_i|$, where $|\alpha_i|$ is the number of statements examined in the $i$th iteration. It follows that a fault-localization technique can also be evaluated in terms of its cumulative effectiveness ($\mathcal{EXAM}$ or "expense") across all iterations.

To further illustrate this via an example, we randomly select a five-bug version[8] (a program with five bugs in it) from among the 75 multibug versions generated for the Siemens suite and show how CBT is applied to locate all the bugs sequentially. For comparison purposes, an evaluation based on Tarantula is also provided. Table XXII presents the data corresponding to this experiment and reports the number of statements examined by either technique, for both the best and worst effectiveness, across each iteration. From Table XXII, we observe that not just cumulatively, but across each and every iteration, CBest is always better or at least equal to TBest, and the same is true when we compare CWorst and TWorst. Furthermore, cumulatively CWorst also performs better than TBest. As per the data corresponding to this five-bug example, we observe that even if we follow a strategy, such as the one-bug-at-a-time strategy, CBT still performs better than Tarantula.

In summary, this section illustrates both the use of the proposed CBT on programs that may contain multiple faults, as well as evaluates CBT against the Tarantula technique using the $\mathcal{EXAM}$ (or "expense") score and the one-bug-at-a-time strategy. As shown, CBT is equally applicable to programs with multiple bugs as well, and our data suggests that it still performs better than Tarantula.

## VIII. DISCUSSION

We now discuss some important concerns related to the CBT and address the threats to validity.

### A. Significance Level $\sigma$

The level of significance $\sigma$ has an impact on how confidently the null hypothesis can be rejected. A hypothesis may be rejected with a significance level of $\sigma = 0.05$ but not when $\sigma = 0.01$. However, in the case of the crosstab-based fault-localization technique, the formula we used to compute the statement suspiciousness does not depend on $\sigma$, i.e., different values of $\sigma$ do not change the statement suspiciousness computed by using (6). As discussed earlier, instead of the so-called "dependence"/"independence" relationship, we are more interested in the degree of association between the execution result and the coverage of each statement. Note that the "dependence" relationship indicates a high association, whereas the "independence" relationship implies a low association. In short, we only use hypothesis testing to make the classification of different statements more understandable (see five classes in Section III-A). The actual ranking of the statements is still only based on their suspiciousness computed as per (6), and therefore, the value of $\sigma$ that is set does not affect the effectiveness of our technique in any way.

### B. Chi-Square Test Versus Fisher's Test

One assumption for using chi-square test is that the sample size cannot be too small. If this is not the case, the Fisher's test should be used to determine whether the null hypothesis should be rejected [7]. For the data in Table II, the Fisher statistic is given by

$$\mathcal{F}(\omega) = \frac{N_S! \times N_F! \times N_C(\omega)! \times N_U(\omega)!}{N_{\mathrm{CS}}(\omega)! \times N_{\mathrm{CF}}(\omega)! \times N_{\mathrm{US}}(\omega)! \times N_{\mathrm{UF}}(\omega)! \times N!} \tag{7}$$

where $N! = N \times (N-1) \times \cdots \times 2 \times 1$. The Fisher's test uses the exact probability distribution of the observed $N_{\mathrm{CS}}(\omega)$, $N_{\mathrm{CF}}(\omega)$, $N_{\mathrm{US}}(\omega)$, and $N_{\mathrm{UF}}(\omega)$, whereas the chi-square test uses the chi-square distribution to approximate the distribution of the statistic $\chi^2(\omega)$. In this paper, we use the chi-square test instead of the Fisher's test for the following reasons. First, for a large sample size (which is true of all our experiments), the distribution of $\chi^2(\omega)$ is a close approximation of a chi-square

[8]In the interests of promoting repeatability of experiments and reproducibility of results, the five-bug version that we used is a version of print_tokens2 that combines the faults in the first, fourth, sixth, seventh, and 10th faulty versions into a single faulty version with all five bugs in it.

distribution. Second, the computational complexity of $\chi^2(\omega)$ [see (1)] is much less than that of $\mathcal{F}(\omega)$ [see (7)].

### C. How Crosstab and Tarantula Work Differently

For a statement $\omega$, its suspiciousness by Crosstab and Tarantula are computed by (6) and (8), respectively,

$$\tau(\omega) = \frac{X}{X+Y} \tag{8}$$

where $X =$ (number of failed tests that execute $\omega$)/(total number of failed tests) and $Y =$ (number of successful tests that execute $\omega$)/(total number of successful tests). From (1) [which is used for the suspiciousness computation in (6)], for a $2 \times 2$ crosstab, we have

$$\chi^2(\omega) = \frac{N^3(N_F \times N_{\mathrm{CS}}(\omega) - N_S \times N_{\mathrm{CF}}(\omega))^2}{N_F \times N_S \times N_C(\omega) \times N_U(\omega)}. \tag{9}$$

Let $N_F = a\ N_S$ and $N_{\mathrm{CF}}(\omega) = b(\omega)\ N_{\mathrm{CS}}(\omega)$, where $a$ is a constant and $b$ may change for different statements. Then, we have $\tau(\omega) = 1/\left(1 + (a/b(\omega))\right)$ and $\chi^2(\omega) = \left(N^3(a - b(\omega))^2\right) / \left(a(b(\omega)+1)(N/N_{\mathrm{CS}}(\omega)) - (b(\omega)+1)\right))$. We observe that $\tau(\omega)$ is determined only by $b(\omega)$, but $\chi^2(\omega)$ is determined by both $b(\omega)$ and $N_{\mathrm{CS}}(\omega)$. For any two statements $\omega_1$ and $\omega_2$, if $\tau(\omega_1) < \tau(\omega_2)$, this implies $b(\omega_1) < b(\omega_2)$. However, this does not guarantee that $\zeta(\omega_1) < \zeta(\omega_2)$ because $\chi^2(\omega)$ also depends on $N_{\mathrm{CS}}(\omega)$, and we do not know whether $\varphi(\omega) < 1$ or $\varphi(\omega) > 1$, which decides whether $\zeta(\omega)$ is positive or negative [as per (6)]. See the example in Section III-B, for $s_6$ and $s_9$, we have $\tau(s_6) = (4/9)/\left((4/9) + (10/27)\right) = 0.5455$ and $\tau(s_9) = (7/9)/((7/9) + (19/27)) = 0.5250$ which implies $\tau(s_9) < \tau(s_6)$. However, for the same statements, we have $\zeta(\omega_6) = 0.1558$ and $\zeta(\omega_9) = 0.1846$, which indicates $\zeta(\omega_9) > \zeta(\omega_6)$. This shows that $\tau(\omega_1) < \tau(\omega_2)$ does not necessarily imply $\zeta(\omega_1) < \zeta(\omega_2)$. Stated differently, if a statement has a smaller suspiciousness than another statement based on Tarantula, it does not necessarily imply that the same also holds for crosstab-based suspiciousness. As a result, statement-rankings using Tarantula can be different from that using CBT, and consequently, the effectiveness of these two techniques will also be different. This is why in Section IV, whenever a comparison between CBT and Tarantula is made in terms of the number of statements that require examination to locate all of the faults, the comparison is on the number and not on the sets of statements themselves, i.e., there may not be any subset/superset relationship between the statements examined by CBT, and those by Tarantula. Data from our case studies suggest that generally, our CBT is more effective than Tarantula in terms of the number of statements that need to be examined.

### D. Clustering-Based Strategies for Programs With Multiple Bugs

In contrast with the one-bug-at-a-time strategy discussed in Section VII, an alternative approach discussed by Jones *et al.* [13] is to construct *fault-focused clusters* by grouping failed tests together based on which fault(s) they are related to, and then, use these clusters to localize specific faults simultaneously. Thus, fault localization for programs with multiple bugs becomes a two-step process as follows.

1) Group test cases that fail on a multibug program into fault-focused clusters such that those in the same cluster are related to the same bug.
2) Combine the failed test cases in each cluster with the successful test cases, and apply fault-localization techniques, such as Tarantula and CBT to locate the fault the cluster is related to.

In terms of achieving such a clustering of failed executions, Podgurski *et al.* [23] analyze execution profiles of failed test cases and correlate each of them with program bugs. However, different failed tests even with respect to the same bug can have very different execution profiles. Since their clustering is based on the similarity between execution profiles, the result does not necessarily imply an accurate causation relationship between certain faults and failed executions. This limitation may degrade its fault-localization capability.

Liu and Han [19] propose to cluster failed execution traces that suggest roughly the same fault location(s). They define the corresponding clustering metric R-PROXIMITY to measure the distance between failed traces. R-PROXIMITY is computed by first using SOBER [18] to generate a predicate ranking for each test case, and then, calculation of the agreement between the different rankings using a weighted Kendall tau distance. Since failed traces are clustered based on the fault locations that they suggest, the results can provide clues as to the locations of the faults associated with each cluster.

In contrast, this paper and the work proposed herein focuses on proposing and evaluating a fault-localization technique. Investigating clustering approaches and evaluating which approach provides better fault-focused clusters than others is beyond the scope of this paper, and undesirably shifts the focus off of CBT. This is why the one-bug-at-a-time strategy has been used to evaluate CBT and Tarantula for programs with multiple bugs (as per Section VII), and we defer our investigation of the use of clustering techniques in the context of fault localization for multibug programs to future work.

### E. Threats to Validity

There may be several threats to the validity of this paper that include, but are not limited to, the following.

An important threat to the construct validity concerns the use of the $\mathcal{EXAM}$ score as a measure of the effort spent by a programmer to localize the faults. We note that this does not provide a complete picture of effort spent, as programmers rarely examine statements one at a time and may spend varying amounts of time examining different statements. In addition, for the results presented in this paper, we assume that if a programmer examines a faulty statement, they will identify the corresponding fault. At the same time, a programmer will not identify a nonfaulty statement as faulty. If such *perfect bug detection* does not hold, then the number of statements that need to be examined to identify a bug may increase. However, such a concern also applies to other fault-localization techniques, and therefore, this is a common limitation.

As explained in Section IV, not every faulty version downloaded is usable by us. For example, if a faulty version does not fail on any test case, then it is excluded from our experiments. Additional faulty versions are generated using mutation-based fault injection, which has shown to be an effective approach to simulate realistic faults that can be used in software testing research to yield trustworthy results [3], [6], [20]. Other researchers such as those of [18] also argue that although these faults are manually injected, they do resemble common logic errors. Therefore, we believe that the seeding of faults in our experiments was a necessary action and does not take away from the correctness of our results in any way. Seeding more faults allowed us to evaluate CBT against a larger number of faulty versions, which is essential for such an empirical study. A complete list of all injected faults for each of the studied programs is available upon request. Examples of the mutant operators (also referred to as "mutation operators" in the literature) used by us to create faulty versions are "relational operator replacement" (ROR) operator, "logical operator replacement" (LCR) operator, etc., [21], [33].

As far as threats to external validity are concerned, it can be argued by some that the evaluation of the effectiveness of CBT is based on empirical data, and therefore, our results may not be generalized for all programs. We do not argue with this but rather emphasize that this is precisely the reason we decided to observe the effectiveness of CBT across a large number of programs. Furthermore, we decided to extend our experiments to not just include C programs but also a Java program. Thus, our study encompasses 22 programs—the Siemens suite, the Unix suite, space, grep, gzip, make, and Ant. Each of these programs varies in its size, purpose, number of faulty versions, number of test cases, etc., which reinforces our belief in the applicability of CBT to different programs.

## IX. Conclusion and Future Work

This paper presents a crosstab-based statistical fault-localization technique (CBT) that uses the coverage information of each executable statement (in terms of how many successful tests and how many failed tests cover each statement, respectively) and the execution result (success or failure) with respect to each test case. Our CBT has been evaluated with respect to 22 programs (both C and Java)—the Siemens suite, the Unix suite, space, grep, gzip, make, and Ant—and results clearly demonstrate its superiority over the Tarantula fault-localization technique. A comparison has also been made between CBT and two other statistical fault-localization techniques—Liblit05 and SOBER, and with respect to our data on the Siemens suite, we find that CBT performs better than them. CBT has also been evaluated for its computational efficiency in terms of time, and results indicate that CBT offers significant improvement over Tarantula in terms of its fault-localization effectiveness without sacrificing computational efficiency. Additional experiments have been performed to investigate the quality of fault localization when large comprehensive test sets are unavailable, and reveal that the crosstab is not extremely sensitive to test set size and still performs well. Experiments are also performed to demonstrate how the crosstab-based fault-localization technique may easily be applied to programs with multiple bugs as well.

Our future work includes, but is not limited to, performing even more experiments on small and large-sized programs, applying the CBT to industrial software that is in ongoing development, and further enhancing the fault-localization process for programs with multiple bugs. We also wish to assess the impact of different coverage types, and different fault types, on the quality of fault localization using our CBT.

## Acknowledgment

The authors would like to thank T. Wei and Y. Qi for assisting in the collection of the data required for the experiments and providing valuable insights and comments while this paper was a work in progress. They would also like to thank Telcordia Technologies (formerly Bell Communications Research, also known as Bellcore) and Atlassian Pty. Ltd., for providing us with the tools—$\chi$Suds and Clover.

## References

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, 2009.

[2] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong, "Fault localization using execution slices and dataflow tests," in *Proc. 6th IEEE Int. Symp. Softw. Rel. Eng.*, Toulouse, France, Oct.1995, pp. 143–151.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proc. 27th Int. Conf. Softw. Eng.*, St. Louis, MI, May 2005, pp. 402–411.

[4] A. Cancellieri and A. Giorgi, "Array preprocessor user manual," in *Interfaces to Database Systems*, 1994.

[5] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proc. 27th Int. Conf. Softw. Eng.*, St. Louis, MI, May 2005, pp. 342–351.

[6] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 733–752, Sep. 2006.

[7] B. S. Everitt, *The Analysis of Contingency Tables*. London, U.K.: Chapman & Hall, 1977.

[8] D. Freeman, *Applied Categorical Data Analysis*. New York: Marcel Dekker, 1987.

[9] L. A. Goodman, *The Analysis of Cross-Classification Data having Ordered Categories*. Cambridge, MA: Harvard Univ. Press, 1984.

[10] http://sir.unl.edu/portal/index.html (Software-artifact Infrastructure Repository), 2011.

[11] http://www.atlassian.com/software/clover/ (Clover: A code coverage analysis tool for Java), 2011.

[12] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proc. 16th Int. Conf. Softw. Eng.*, Sorrento, Italy, May 1994, pp. 191–200.

[13] J. A. Jones, J. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proc. Int. Symp. Softw. Testing Anal.*, London, U.K., Jul. 2007, pp. 16–26.

[14] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proc. 20th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Long Beach, CA, Nov. 2005, pp. 273–282.

[15] M. Kendall and A. Stuart, *The Advanced Theory of Statistics*. London, U.K.: Griffin, 1979.

[16] B. Liblit, A. Aiken, A. Zheng, and M. Jordan, "Bug isolation via remote program sampling," in *Proc. ACM SIGPLAN Int. Conf. Program. Language Design Implement.*, 2003, pp. 141–154.

[17] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proc. ACM SIGPLAN Conf. Program. Language Design Implement.*, Chicago, IL, Jun. 2005, pp. 15–26.

[18] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: a hypothesis testing-based approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 10, pp. 831–848, Oct. 2006.

[19] C. Liu and J. Han, "Failure proximity: a fault localization-based approach," in *Proc. 14th ACM SIGSOFT Symp. Foundations Softw. Eng.*, Portland, OR, Nov. 2006, pp. 286–295.
[20] A. S. Namin, J. H. Andrews, and Y. Labiche, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, Aug. 2006.
[21] J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 2, pp. 99–118, Apr. 1996.
[22] R. Lyman Ott, *An Introduction to Statistical Methods and Data Analysis*, 4th ed. Belmont, CA: Duxbury/Wadsworth, 1993.
[23] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Proc. 25th Int. Conf. Softw. Eng.*, Portland, OR, May 2003, pp. 465–475.
[24] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proc. 18th IEEE Int. Conf. Automated Softw. Eng.*, Montreal, QC, Canada, Oct. 2003, pp. 30–39.
[25] R. Santelices, J. A. Jones, Y. Yu, and M. J. Harrold, "Lightweight fault-localization using multiple coverage types," in *Proc. 31st Int. Conf. Softw. Eng.*, Vancouver, Canada, May, 2009, pp. 56–66.
[26] The Siemens Suite. http://www-static.cc.gatech.edu/aristotle/Tools/subjects/, Jan. 2007.
[27] I. Vessey, "Expertise in debugging computer programs," *Int. J. Man-Machine Stud.: Process Anal.*, vol. 23, no. 5, pp. 459–494, 1985.
[28] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," *Softw.: Practice Experience*, vol. 28, no. 4, pp. 347–369, Apr. 1998.
[29] W. E. Wong and Y. Qi, "Effective program debugging based on execution slices and inter-block data dependency," *J. Syst. Softw.*, vol. 79, no. 7, pp. 891–903, Jul. 2006.
[30] W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *J. Syst. Softw.*, vol. 83, no. 2, pp. 188–208, Feb. 2010.
[31] W. E. Wong and Y. Qi, "BP neural network-based effective fault localization," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 19, no. 4, pp. 573–597, Jun. 2009.
[32] W. E. Wong, Y. Shi, Y. Qi, and R. Golden, "Using an RBF neural network to locate program bugs," in *Proc. 19th IEEE Int. Symp. Softw. Rel. Eng.*, Seattle, WA, Nov. 2008, pp. 27–38.
[33] W. E. Wong and A. P. Mathur, "Reducing the cost of mutation testing: An empirical study," *J. Syst. Softw.*, vol. 31, no. 3, pp. 185–196, Dec. 1995.
[34] W. E. Wong, T. Wei, Y. Qi, and L. Zhao, "A crosstab-based statistical method for effective fault localization," in *Proc. 1st Int. Conf. Softw. Testing, Verification Validation*, Lillehammer, Norway, Apr. 2008, pp. 42–51.
[35] $\chi$Suds User's Manual, Telcordia Technol., 1998.
[36] Y. Yu, J. A. Jones, and M. J. Harrold, "An empirical study on the effects of test-suite reduction on fault localization," in *Proc. Int. Conf. Softw. Eng.*, Leipzig, Germany, May 2008, pp. 201–210.
[37] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proc. 10th ACM SIGSOFT Symp. Foundations Softw. Eng.*, Charleston, SC, Nov. 2002, pp. 1–10.
[38] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 183–200, Feb. 2002.
[39] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang, "Capturing propagation of infected program states," in *Proc. 7th Joint Meeting. Eur. Softw Eng. Conf. ACM SIGSOFT Symp. Foundations Softw. Eng.*, Amsterdam, The Netherlands, Aug. 2009, pp. 43–52.
[40] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proc. 28th Int. Conf. Softw. Eng.*, Shanghai, China, May 2006, pp. 272–281.
[41] A. Zheng, M. Jordan, B. Liblit, M. Naik, and A. Aiken, "Statistical debugging: simultaneous isolation of multiple bugs," in *Proc. 23rd Int. Conf. Mach. Learning*, Pittsburgh, PA, Jun. 2006, pp. 1105–1112.

**W. Eric Wong** received the Ph.D. degree in computer science from Purdue University, West Lafayette, IN.

He is currently an Associate Professor in computer science with the University of Texas, Dallas (UTD). Prior to joining UTD, he was with Telcordia (formerly Bellcore) as a Project Manager for Dependable Telecom Software Development.

Dr. Wong received the Quality Assurance Special Achievement Award from Johnson Space Center, National Aeronautics and Space Administration, in 1997. He was also the Secretary of the Association for Computing Machinery Special Interest Group on Applied Computing. He is on the Administrative Committee of the IEEE Reliability Society.

**Vidroha Debroy** (S'08) received the B.S. degree in software engineering and the M.S. degree in computer science from the University of Texas, Dallas, where he is currently working toward Ph.D. degree in computer science.

His current research interests include software testing and fault localization, program debugging, and automated and semiautomated ways to repair software faults.

**Dianxiang Xu** received the Ph.D. degree in computer science from Nanjing University, Nanjing, China.

He is currently an Associate Professor with the National Center for the Protection of the Financial Infrastructure, Dakota State University, Fargo. Madison, SD. From 2003 to 2009, he was an Assistant Professor of computer science at North Dakota State University, Fargo. His research interests include the areas of software security and safety, software testing, applied formal methods, and computer forensics.