Fault Localization with Non-Parametric Program Behavior Model *

Peifeng Hu
The University of Hong Kong
Pokfulam, Hong Kong
pfhu@cs.hku.hk

Zhenyu Zhang
The University of Hong Kong
Pokfulam, Hong Kong
zyzhang@cs.hku.hk

W. K. Chan
City University of Hong Kong
Tat Chee Avenue, Hong Kong
wkchan@cs.cityu.edu.hk

T. H. Tse †
The University of Hong Kong
Pokfulam, Hong Kong
thtse@cs.hku.hk

Abstract

Fault localization is a major activity in software debugging. Many existing statistical fault localization techniques compare feature spectra of successful and failed runs. Some approaches, such as SOBER, test the similarity of the feature spectra through parametric self-proposed hypothesis testing models. Our finding shows, however, that the assumption on feature spectra forming known distributions is not well-supported by empirical data. Instead, having a simple, robust, and explanatory model is an essential move toward establishing a debugging theory. This paper proposes a non-parametric approach to measuring the similarity of the feature spectra of successful and failed runs, and picks a general hypothesis testing model, namely the Mann-Whitney test, as the core. The empirical results on the Siemens suite show that our technique can outperform existing predicate-based statistical fault localization techniques in locating faulty statements.
Keywords: Fault localization, non-parameter statistics

1. Introduction

Software debugging is time-consuming and is often a bottleneck in the software development process. It involves at least two crucial steps, namely *fault localization* and *fault correction*. Fault localization identifies the causes of abnormal behaviors of a faulty program. Fault correction modifies the faulty program or data structure to eliminate the effect of the identified faults.

A traditional fault localization technique consists of setting breakpoints, re-executing the faulty program on the inputs, and examining the corresponding program states. Recently, statistical fault localization techniques [8, 10, 12–15] are proposed and reported to be promising. They locate faults by analyzing the statistics of dynamic program behaviors. A *failed run* is a program execution that reveals a failure, and a *successful run* is a program execution that reveals no failure. A statistical fault localization technique locates a fault-relevant statement (or a faulty statement directly) by comparing the statistical information of program elements in these two kinds of run. Such program elements can be statements [10] or predicates [12, 13]. Liu et al. [15], for instance, report that statistical techniques can achieve more accurate results than other approaches. Previous experiments [12–15] show that such techniques can identify about 2 faults out of 3 by examining 20% of all code.

Because of their statistical nature, the techniques assume that there are statistically enough successful and failed runs to collectively help locate faults. These techniques build underlying statistical behavior models for the aggregated execution data of selected program elements (known as *features*), and search for elements that correlate with the observed program failures.

* This research is supported in part by a grant of the Research Grants Council of Hong Kong (project nos. 111107, 123207, and 716507).

† All correspondence should be addressed to Prof. T. H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2183. Fax: (+852) 2858 4141. Email: thtse@cs.hku.hk.

1550-6002/08 $25.00 © 2008 IEEE
DOI 10.1109/QSIC.2008.44

To compare the spectra of features, there are diverse approaches. Tarantula [8, 10] gauges the fraction (x) of successful runs and the fraction (y) of failed runs with respect to the execution of a statement. It then uses the ratio $\frac{1}{y/x+1}$ to predict how much the statement is correlated to the observed failure. It also develops a strategy to rank statements according to the relative magnitude of the ratio associated with each statement.

Instead of locating the faulty statements directly, predicate-based statistical techniques, such as CBI [12, 13] and SOBER [14, 15], locate the program predicates related to faults. A *program predicate* is a Boolean expression about the property of a system at some program location (such as a statement). CBI [12, 13] and SOBER [14, 15] contrast the feature spectra of predicates in a program. They collect statistics about the behaviors of program predicates, such as evaluations of branch statements. They further assume that, for predicates near the fault position, the successes or failures of their evaluations are highly correlated to the successes or failures of program executions. Hence, identifying effective program predicates and formulating correct and robust statistic comparisons are important for such techniques.

CBI [12, 13] checks the probability of a predicate to be evaluated to be true in all failed runs and that in all the runs (irrespectively of whether successful or failed), and measures the *increase* from the former to the latter. This increase is used as the ranking score, which indicates how much the predicate is related to a fault. SOBER [14, 15] defines *evaluation bias* to estimate the chance that a predicate is evaluated to be true in each run. More precisely, if P is a predicate and $\pi(P)$ is the probability that it is evaluated to be true in every run, then $\pi(P)$ can be evaluated by $\frac{n_t}{n_t+n_f}$, where n_t is the number of times that P is evaluated to be true and n_f is the number of times that P is evaluated to be false. SOBER then evaluates the difference between the distributions of $\pi(P)$ for successful and failed runs, and deems that the larger the difference, the more will P be relevant to a fault. In brief, CBI and SOBER use similar kinds of statistical mean comparison.

However, the above techniques have a couple of limitations: Firstly, Tarantula and CBI do not distinguish the number of times that a particular program element (statement or predicate) has been executed in a run. Liu et al. [15] empirically show that such a method can be less accurate than one in which the distributions of evaluation biases assembled from successful and failed runs are considered. Our study in this paper shows similar results. Secondly, SOBER uses the central limit theorem in statistics to measure the behavioral difference of a predicate for successful and failed runs. In the implementation provided in the authors' website, it implicitly assumes that $\pi(P)$ is normally distributed. Our empirical study on the Siemens suite [5] shows that most predicates are far from having any known distribution. Hence, a parametric hypothesis testing approach may lose its discrimination capability significantly and produce non-robust results.

These motivate us to adopt a statistical fault localization approach and a generic model without the undesirable effects. In particular, we propose to use a non-parametric approach. To evaluate the work, we stipulate our model in the context of predicate-based statistical fault localization, and pick a popular non-parametric hypothesis testing, the Mann-Whitney test, to determine the degree of difference between the spectra of program elements for successful and failed runs. The degree of difference can be used as the ranking score, which indicates how much a predicate is related to a fault. Based on the ranking scores of the predicates, we obtain a ranked list of predicates. Predicates having high ranking scores are deemed to be suspicious. Debuggers may use the suspicious predicates to start the search for program faults. Empirical results show that our method is effective in locating faults in programs.

The main contributions of the paper are two-fold: (i) It gives the first case study on the statistical nature of the execution spectra over the Siemens suite. It shows that certain features do not follow a Gaussian or normal distribution. Such a finding highlights a threat to construct validity of the empirical results reported in many fault localization experiments in the literature. It also serves a reference point for fellow researchers to work on statistical approaches that mine software execution data from program behaviors. (ii) It demonstrates the use of a non-parametric hypothesis testing to improve the robustness of existing fault-relevant predicate ranking models. Our experiments show that our model can discover 70% of the faults when examining up to 20% of the code.

The remainder of the paper is organized as follows. Section 2 provides a motivating study. We then discuss our approach in Section 3, after which we describe our empirical evaluation in Section 4. Related work is presented in Section 5. Section 6 concludes the paper.

2. Motivating Study

In this section, we use a program from the Siemens suite [5] to illustrate our important initial finding on the statistics of program behaviors. Figure 1 shows the

```
P1:     if ( rdf ≤ 0 || cdf ≤ 0 ) {
          info = -3.0;
          goto ret3;
        }
        ⋮
P2:     for ( i = 0; i < r; ++i ) {
          double sum = 0.0;
P3:       for ( j = 0; j < c; ++j ) {
            long k = x(i,j);
P4:         if ( k < 0L ){
              info = -2.0;
E1:           /*goto ret1;*/
            }
            sum += (double)k;
          }
          N += xi[i] = sum;
        }
P5:     if ( N ≤ 0.0 ) {
          info = -1.0;
          goto ret1;
        }
P6:     for ( j = 0; j < c; ++j ) {
          double sum = 0.0;
P7:       for ( i = 0; i < r; ++i )
            sum += (double)x(i,j);
          xj[j] = sum;
        }
        ⋮
        ret1:
```

Figure 1. Excerpt from faulty version 1 of program "tot_info" from the Siemens suite.

code excerpted from faulty version 1 of the program "tot_info". Seven predicates are included, labeled as P_1 to P_7. The statement "goto ret1;" (labeled as E_1) is intentionally commented out by the Siemens researchers to simulate a statement omission fault. Locating such a kind of fault is often difficult even if the execution of a failed test case is traced step-by-step.

Let us focus on program behaviors resulting from predicate evaluations because they have been successfully used in fault localization research such as SOBER. We observe that the predicate "P_4: k < 0L" is highly relevant to program failures because the omitted statement E_1 is in the *true* block of the branch statement. We further find that none of the predicates P_1, P_2, and P_3 is related to failures because they neither directly activate the fault nor propagate an error. Predicate P_5 is also related to the fault, since commenting out the goto statement (E_1) will render a higher chance for P_5 to be evaluated. Predicates P_6 and P_7 are increasingly distant from the faulty statement E_1.

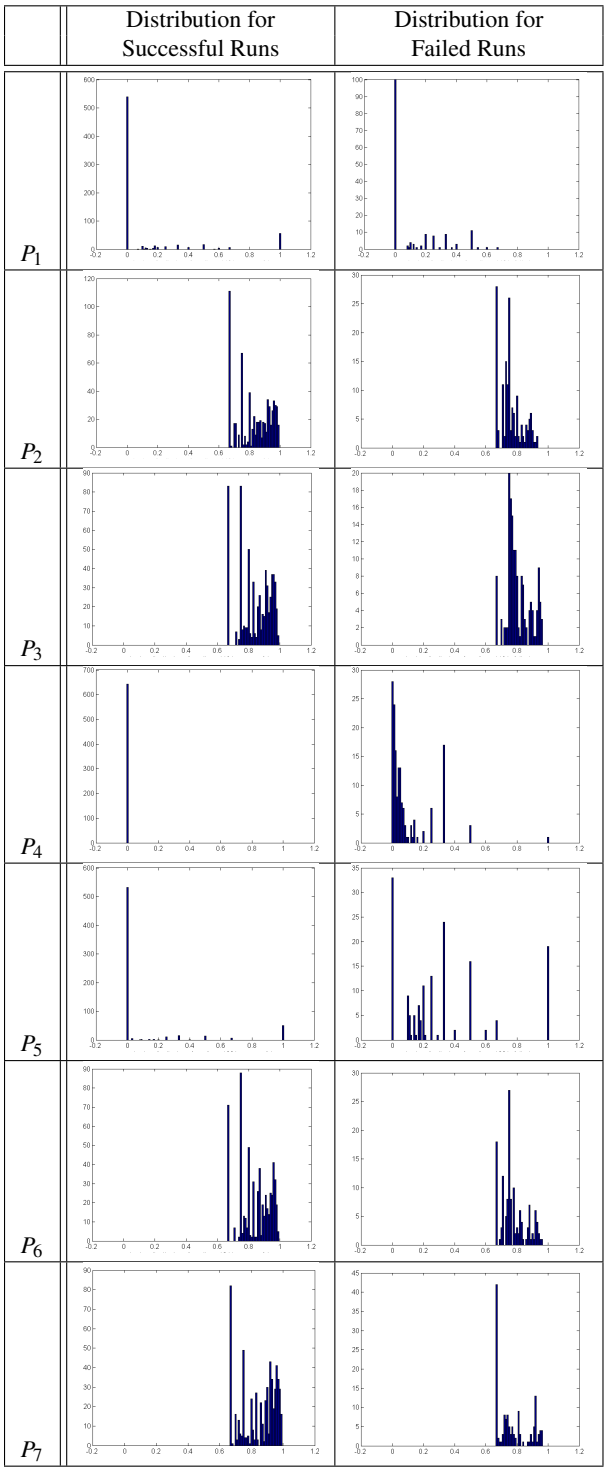
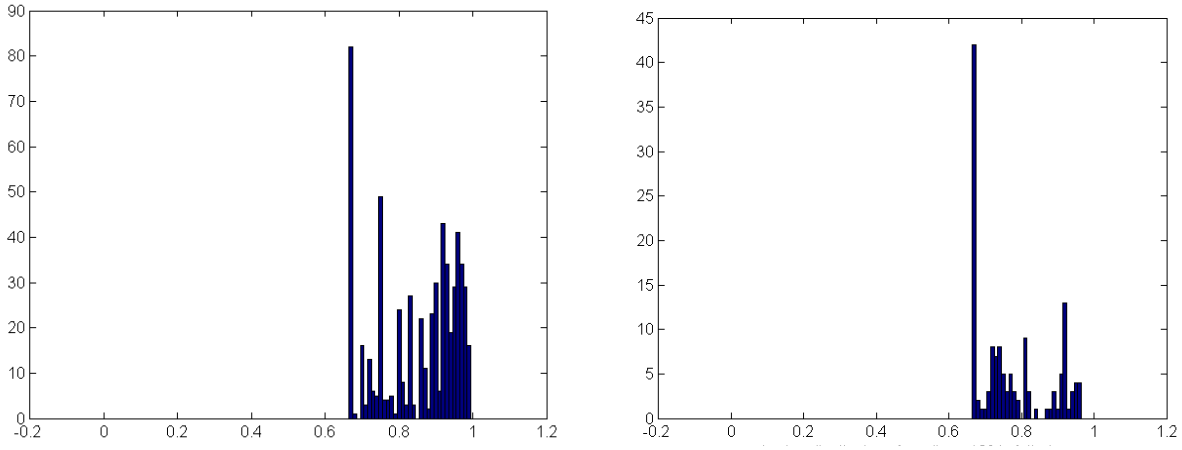


Figure 2. Distributions of evaluation biases for predicates P_1 to P_7.



(a) Distribution for successful runs

(b) Distribution for failed runs

Figure 3. Distributions of evaluation biases for predicate P_7.

The 7 pairs of distributions of evaluation biases with respect to P_1 to P_7 are shown via the histograms in Figure 2. We also zoom in the histograms for P_7 in Figure 3.

In each of these plots, the X-axis stands for the evaluation biases (varying in the range of $[0, 1]$), and the Y-axis is the number of (successful or failed) runs that share the same value of evaluation bias. They are produced by executing the program over all the test cases in Siemens suite. If a predicate is not executed in a run, there will be no data captured in the distribution.

The series of histograms (distributions of evaluation biases) on the left are for the successful runs and those on the right are for the failed runs. The resolution (step) of the histograms is 0.01. Take the plot in Figure 3(a) as an illustration. The left-most bar means that there are 82 successful test cases, over which the evaluation biases of P_7 in their corresponding program execution are in the range of $[0.65, 0.66)$.

We have the following observations from the histograms:

*O*1: Evaluation biases for predicates (see Definition 1 in Section 3.1) are not always close to 0 or 1, but are scattered throughout the range of $[0, 1]$. It means that simply checking whether it is evaluated to be true or false may lead to information loss and inaccurate analyses.

*O*2: The histograms for predicate P_1 resemble each other. The same phenomenon is observed for P_2, P_3, P_6, and P_7. Those for P_4 and P_5, however, differ significantly. It indicates that the differences of distributions over successful and failed runs can be good indicators of the fault-relevance of predicates.

*O*3: None of the histograms in Figures 2 and 3 resembles a Gaussian or normal distribution. For each predicate of every program in the Siemens suite, we have conducted the standard t-test to determine whether its evaluation bias follows a Gaussian distribution. The results show that for nearly 60% of a total of 10042 distributions, the assumption of Gaussian distribution is rejected at the 5% significance level. We further observe that, as far as the programs under study can represent, assuming a normal distribution for predicate evaluation bias is unrealistic.

Besides, SOBER sets the evaluation bias of a predicate for a successful run to be 0.5 if the predicate is never evaluated in the run. Is this a valid assumption? We observe that the values in Figure 2 are not always distributed across the entire range of $[0, 1]$. For predicate P_7, for instance, the values only lie within the range of $[0.67, 1]$. In other words, the actual evaluation bias for any run *cannot* take the value of 0.5. In fact, this situation is not an isolated case. Indeed, out of all the 142 instrumented predicates in the program "tot_info", the ranges of evaluation biases for 124 of them (or 87.3%) never include the value of 0.5.

Unlike the reasoning in [15], we argue that it is not a fair assumption to set 0.5 as the value of evaluation bias for predicates not evaluated in a run. In fact, any artificial value has a similar problem. This motivates us to abandon the use of any preset value (including 0.5) in our model. Furthermore, in practice, there often exist only small percentages of test

cases that reveal failures, and the pool of test cases is usually not large. A parametric hypothesis testing technique or the central limit theorem is *not* suitable for non-parametric distributions with small samples. Mathematicians have proposed many non-parametric analysis techniques instead.

In summary, to conduct statistical fault localization, we propose to use generic non-parametric analysis techniques to compare the statistics from dynamic program behaviors.

3. Our Predicate Ranking Model

In this section, we explore a non-parametric model for ranking fault-relevant program locations by fully utilizing the statistical distribution information from successful and failed runs.

3.1. Preliminaries

We first revisit the notion of program predicates and evaluation biases [13, 15].

Liblit et al. [12, 13] propose three types of program location to sample the execution statistics of successful and failed runs. Each program location is associated with a set of Boolean predicates. Collectively, they define the set of program predicates in the program. The three types of program location are as follows:

- *Branches*: At each conditional statement, CBI tracks the conditional *true* and *false* branches via a pair of program predicates, which monitor whether the corresponding branches have been taken. SOBER further collects the number of times that the branches have been taken in a run.

- *Returns*: At each return statement, six predicates are tracked to find whether the returned value r satisfies $r < 0$, $r \leq 0$, $r > 0$, $r \geq 0$, $r = 0$, and $r \neq 0$, respectively. Both CBI and SOBER collect evaluation biases for these predicates.

- *Scalar-pairs*: To track the relationship between a variable and another variable or constant in each assignment statement, six predicates (similar to those for return statements above) are adopted by CBI. On the other hand, SOBER experimentally verifies and concludes that *not* tracking these predicates will not degrade the fault localization quality when using the Siemens suite.

Each program predicate may be executed more than once in a run. Each evaluation will give either a true or a false value. We thus give the notion of evaluation bias to estimate the probability of a predicate being evaluated as true in a run as follows:

Definition 1 (Evaluation Bias [15]) *Let n_t be the number of times that a predicate P has been evaluated to be true in a run, and n_f the number of times that it has been evaluated to be false in the same run. $\pi(P) = \frac{n_t}{n_t+n_f}$ is called* **evaluation bias** *of predicate P in this particular run.*

Intuitively, some predicates may not be evaluated in a run. SOBER sets 0.5 as their evaluation biases. In our model, they are not assigned any artificial evaluation bias.

In the next section, we shall elaborate on our non-parametric hypothesis ranking model.

3.2. Non-Parametric Hypothesis Ranking Model

Following the conventions from standard statistics, we treat each run as an independent event. Let T_s and T_f be the whole sets of possible successful runs and failed runs, respectively. Given a random test case t in T_s or T_f, let X be the random variable representing the evaluation bias of predicate P for the program execution over t. We use $f(X|\theta_s)$ and $f(X|\theta_f)$ to denote the probability density functions of the evaluation biases of predicate P on T_s and T_f, respectively. Ideally, if a predicate is relevant to a fault, $f(X|\theta_s)$ should differ from $f(X|\theta_f)$. Moreover, the larger the difference, the more relevant will be P in relation to the fault.

We define a ranking function

$$R(P) = \mathit{Diff}\big(f(X|\theta_s), f(X|\theta_f)\big) \tag{1}$$

to measure the difference between $f(X|\theta_s)$ and $f(X|\theta_f)$.

Without any prior knowledge of $f(X|\theta_s)$ and $f(X|\theta_f)$, we can only estimate them from the sample set, that is, the test suite attached with the program. Each corresponding run of a test case from the test suite is treated as a sample for the random variable X. In this way, we obtain sample sets for $f(X|\theta_s)$ and $f(X|\theta_f)$, respectively. We deem that the difference between the two sample sets is an approximation of $R(P)$. To measure the difference between the two sample sets, a promising way is to use a parametric hypothesis testing method. However, according to standard statistics textbooks such as [16], a parametric hypothesis testing can be meaningfully applied only if

(a) The two sample sets are independently and randomly drawn from the source population;

(b) The scales of measurement for both sample sets have the properties of an equal interval scale;

(c) The source population(s) can reasonably be assumed to have a known distribution.

In cases where the data from two independent samples fail to meet any of these requirements, it is a well-known advice to use a non-parametric alternative. This is further supported by our empirical study presented in Section 2, which shows that the underlying data populations are indeed far from a known distribution model. Therefore, we propose to use the Mann-Whitney test [16], a non-parametric hypothesis testing technique, to measure the differences in the sampled distributions of evaluation biases. The robustness of non-parametric hypothesis testing frees us from having artificial configuration parameters.

In the rest of the section, we describe how the Mann-Whitney test is applied to our fault localization model.

Problem Settings. Let V_s be the sample set of evaluation biases for a predicate P from m successful runs. Similarly, let V_f be the sample set of evaluation biases for predicate P from n failed runs. Without loss of generality, we assume that $m \geq n$. The goal is to gauge the differences between such V_ss and V_fs, and use the measures to rank the predicates.

Our ranking approach strictly follows the Mann-Whitney test, [1] and consists of two steps. It firstly transforms the two sample sets of evaluation biases to two rank-value sets, and then measures the distance between two rank-value sets. In the sequel, we shall use an example to illustrate the process.

Computing the ranks of sampling values in V_s and V_f. We follow the Mann-Whitney test to compute the rank-values of all sampling values in V_s and V_f. Suppose that there is a predicate P whose sets of evaluation biases are $V_s = \{0.2, 0.3, 0.4\}$ and $V_f = \{0.4, 0.5\}$. For simplicity, we do not explicitly list the test case number for each value instance.

The first step is to construct the union set V of V_s and V_f. There are totally 5 samples for the evaluation biases of P, namely 0.2, 0.3, 0.4, 0.4, and 0.5. We assign a rank to each of them. The smallest value 0.2 is assigned a rank of 1. Similarly, the largest value 0.5 is assigned a rank of 5. The value 0.4 appears twice in the list. The algorithm will assign the average rank 3.5 in both cases. After assigning ranks to the values, we obtain a *rank-value* set $RS = \{1, 2, 3.5, 3.5, 5\}$ for V.

[1] We conjunct that other non-parametric tests may be used instead.

We then map the rank-values back to the corresponding elements in V_s and V_f, thus constructing two new sets R_s and R_f, respectively. In this example, for instance, we obtain $R_s = \{1, 2, 3.5\}$ for V_s and $R_f = \{3.5, 5\}$ for V_f.

Measuring the difference between R_s and R_f. After constructing the two rank-value sets, we then enumerate all the distribution possibilities of rank-values inside them. The two sets may contain different number of elements. The Mann-Whitney test selects m out of $m+n$ possibilities in C^m_{m+n} combinations. For instance, there may be 10 combinations for R_s in the example, namely $\{1, 2\}$, $\{1, 3.5\}$, $\{1, 3.5\}$, $\{1, 5\}$, $\{2, 3.5\}$, $\{2, 3.5\}$, $\{2, 5\}$, $\{3.5, 3.5\}$, $\{3.5, 5\}$, and $\{3.5, 5\}$.

We then proceed to use the ranking function $R(P)$ in the Mann-Whitney test to derive the ranking score for the predicate P. Let K be the number of combinations of rank-values (such as $K = C^m_{m+n} = 10$ in this example). We define the *sum of ranks* S of a rank-value set RS to be the sum of the rank-values of all the elements in RS (that is, $S = \sum_{i \in RS} [\textit{rank-value of } i\,]$). In this example, the values of S for the 10 combinations are 3, 4.5, 4.5, 6, 5.5, 5.5, 7, 7, 8.5, and 8.5, respectively. For instance, for the rank-value set of $\{1, 3.5\}$, the sum of ranks is $1 + 3.5$, which is 4.5.

Let K_l be the number of combinations whose sum of ranks is less than that of R_s. Similarly, let K_h be the number of combinations whose sum of ranks is larger than that of R_s. We then calculate the minimum of K_l/K and K_h/K. It indicates the difference between V_s and V_f. The lower the minimum, the more divergent will be the two sampled distributions. Based on the above, we approximate equation (1) by our ranking function

$$R(P) = -min(K_l/K, K_h/K)$$

The higher the ranking score, the more relevant will be P in relation to the fault. In this way, we rank the predicates in decreasing order of ranking scores. If two predicates happen to have equal ranking scores, they share the same rank in the list. In particular, predicates having a significant difference in terms of the hypothesis testing are suspected to correlate with faults.

4. Experiment

In this section, we present the experiment to validate our technique.

4.1. Subject Programs

The Siemens suite consists of 132 C programs with seeded faults. Each program is a variation of one of seven programs, namely "tcas", "tot_info", "replace", "print_tokens", "print_tokens2", "schedule", and "schedule2", varying in size from 133 to 515 executable lines.[2] Each faulty version is seeded with exactly one fault. We have downloaded these programs from the Software-artifact Infrastructure Repository (SIR) [5] website. Table 1 shows the descriptive statistics of the suite, including the number of faulty versions, the number of executable lines of code, the number of test cases, and the percentage of failed test cases.

4.2. Performance Metrics

Performance metrics are always a central issue in accurate and objective comparisons. To gauge the fault localization quality of our method, we use T-scores as the metric, which was originally proposed by Renieres and Reiss [17] and later adopted by SOBER [14, 15] in reporting the performance of their fault localization techniques. We summarize the measure as follows.

Consider a faulty program together with its program dependence graph $G = (N, E)$, where N is the set of statements and E is the set of (data- and/or control-) dependencies between pairs of related statements. The set of faulty statements are denoted by N_{defect}. The list of suspicious statements (in order of suspiciousness) given by a fault localization technique is denoted by N_{blamed}. Starting from a node in N_{blamed}, developers do a breadth-first search and stop when a node in N_{defect} is reached. The set of nodes examined is denoted by $N_{examined}$. The T-score T is given by

$$T = \frac{|N_{examined}|}{|N|} \times 100\%$$

where $|N|$ is the size of the N in G. In essence, it measures the percentage of source code that needs to be examined in order to find a faulty statement. (In some previous work such as [3, 17], $1 - T$ is used.)

The T-score helps measure the cost of locating a fault. The lower the percentage of code examined, the higher will be the effectiveness of a fault localization technique. In practice, developers may select the top n suspicious statements to start the breadth-first search. Accordingly, the result is named as the top-n T-score result. Developers may also specify an upper bound of code examination (such as 20% in previous work [15]).

[2] We use the tool "David A. Wheeler's SLOCCount" to count the executable statements. It is available at http://www.dwheeler.com/sloccount/.

4.3. Setup of Experiment

Among the 132 programs, two of them come with no failed test cases. This is also reported in previous work [14, 15]. These two versions are excluded because both our method and SOBER need the presence of both successful and failed test cases. To evaluate our method, we follow [14, 15] to use the whole test suite as input to our method and SOBER. Again, following [14, 15], we use branches and returns (see Section 3.1) as program locations for predicates in the experiment.

Our experiment is carried out on a Dell Inspiron 6400 laptop with an Inter Core(TM)2 T5600 @ 1.83GHz stepping 06 CPU. The operating system is Ubuntu 6.06 LTS Linux with kernel version 2.6.15-28-386 (buildd@terranova). The Mann-Whitney test in our experiment is implemented using Matlab 7.0.

4.4. Overall Performance Comparison

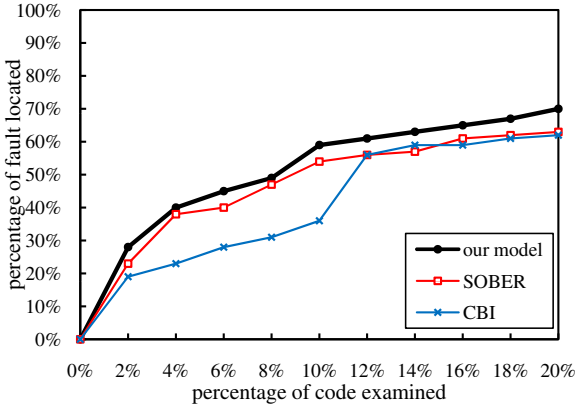


Figure 4. Overall performance comparison of top-5 T-score results.

In this section, we evaluate our model using the Siemens suite and compare our result with those of SOBER and CBI. We note that all the three techniques rank predicates and then produce a list of suspicious faulty predicates from the highest ranking to the lowest ranking.

The results of CBI are directly cited from [15], while the results of SOBER are worked out using our implementation of SOBER and the T-scores according to their papers. Figure 4 depicts the percentage of faults that can be located when a certain percentage of code is examined. We report the results of the top-5 T-score in this section, because [14, 15] report that the top-5 strategy gives SOBER and CBI the best T-score results. We show only the range of $[0\%, 20\%]$, since unlimited

Program	No. of Faulty Versions	Executable LOC	No. of Test Cases	Percentage of Failed Test Cases
print_tokens	7	341–342	4130	1.7%
print_tokens2	10	350–354	4115	5.4%
replace	32	508–515	5542	2.0%
schedule	9	291–294	2650	3.2%
schedule2	10	261–263	2710	1.0%
tcas	41	133–137	1608	2.4%
tot_info	23	272–274	1052	5.6%

Table 1. Statistics of Siemens suite.

code examination is time-consuming and unacceptable, and this range is also used in previous work [14, 15].

Figure 4 shows the aggregated results of our non-parametric testing model, as well as those of SOBER and CBI, on all program versions. Firstly, in the range of $[4\%, 10\%]$, our method obviously outperforms CBI. Generally, we observe that by checking less than 20% code, our method always locate more faults than SOBER or CBI. Take the 10% code examining point for illustration. Out method can locate 59% of all the faults, while SOBER locates 54% and CBI locates 36%, respectively. When a programmer can examine up to 20% of the code, which is conjectured by SOBER to be the upper bound of meaningful code examination that can be afforded, our approach can help discover 70% of all the faults, while SOBER and CBI can only do so for about 63% and 62% of all the faults, respectively. If we deem SOBER and CBI as effective techniques, the result indicates that our method is also effective.

4.5. Individual Performance Comparison

Unlike the overall comparison between CBI and our approach, we do not observe large differences between the results of our model and SOBER in Figure 4. We decide, therefore, to compare their performances on each individual Siemens program.

Figure 5 shows the results of our method and SOBER on each of the seven Siemens programs. The X-axis and Y-axis of the seven plots in Figure 5 can be interpreted similarly to those of Figure 4.

The plots for the seven programs are ordered according to their program sizes, in terms of the executable statement counts as shown in the column "Executable LOC" of Table 1. For the relatively larger programs "replace" and "print_tokens2" (Figures 5(a) and 5(b)), the results of our method consistently outperform those of SOBER. For the relatively smaller program "tcas" (Figure 5(c)), our method produces results comparable with SOBER. For the other programs (Figures 5(d), 5(e), 5(f), and 5(g)), it is difficult to tell which one is better. Our method seems to have better results than SOBER over the programs "tot_info" and "schedule2". However, SOBER catches up with our method over the program "print_tokens". For the program "schedule", neither our method nor SOBER has an obvious advantage over the other.

The results show that, the larger the program scale, the better will be the results of our method when compared to those of SOBER. This is understandable for the following reason: Large programs tend to have many predicates, so that it is harder to execute all of them in a run. For predicated that are not executed, their evaluation biases are set to a value of 0.5 in SOBER. When there are many predicates not executed in a run, there will be many 0.5 entries in the distributions of evaluation biases. They will cause the distributions for successful and failed runs to appear more similar. These entries have been eliminated in our approach. As such, intuitively, our approach is more robust (because of the use of non-parametric test) and more scalable (because of the above elimination).

4.6. Threats to Validity

Internal validity is mainly caused by factors that affect experimental results. To avoid implementation errors, we have implemented SOBER and T-score strictly according to previous work and tested our platform with great care.

Since we use Linux as our experimental environment, the execution statistics of test cases may differ from previous work owing to platform dependencies (which is also explained in [5]). We have manually examined the differences carefully.

Construct validity lies in the evaluation method we choose. Since T-score is widely used in previous work (including [14, 15]), we also use T-score to compare our method with SOBER and CBI. Nevertheless, some limitations have been reported in the use of T-score (see [3] for example). Has any other measures been used to evaluate predicate-based techniques successfully? We are not aware such alternatives in the public literature.

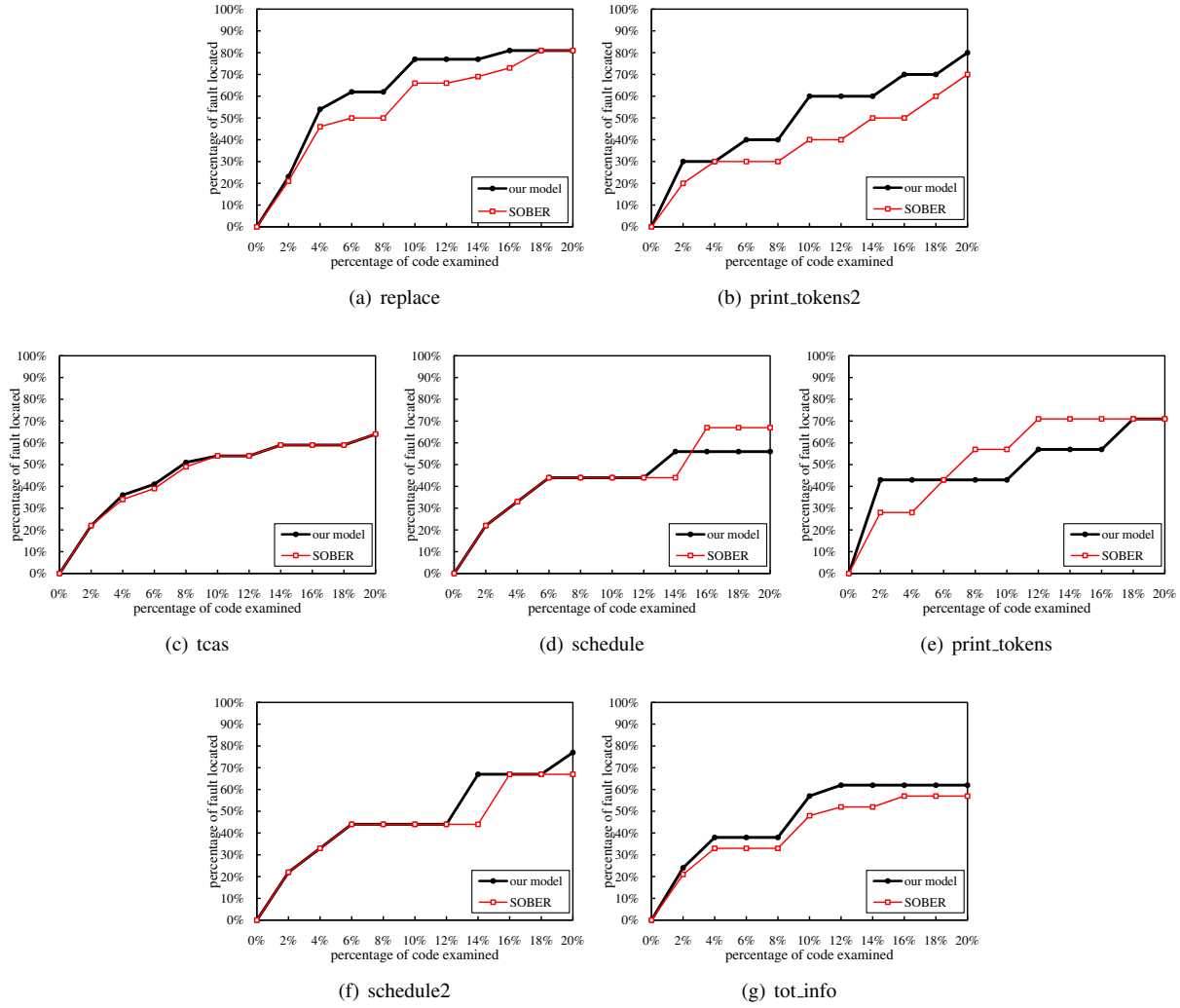


Figure 5. Individual performance comparisons of top-5 T-score results.

On the other hand, it may be unfair to use the same performance measure (T-score) to compare with other techniques (such as [3, 8, 10]) and may lead to unreliable results. We, therefore, adopt only SOBER and CBI as peer techniques and compare our method with them.

Another threat is the predicates we choose to investigate. SOBER has reported that scalar-pair predicates have minor effects on fault localization results. Hence, we follow SOBER to adopt the other two kinds of predicate in the experiment.

External validity is related to the target programs used. Since the faults in Siemens programs are manually seeded, they may not truly represent realistic faults. Using other programs may give different results. More evaluation should, therefore, be done in the future.

5. Related Work

Program slicing [19] is a code-based technique. It is widely used in debugging [18]. Gupta et al. [6] propose a forward dynamic slicing approach to narrow down slices. They further integrate the forward approach with standard dynamic slicing approaches [22].

Collofello and Cousins [4] pioneer the use of test cases for fault localization. A promising approach is to use the behavioral statistics collected from test case executions. *Delta debugging* helps to simplify or iron out fragments of failed test cases [21], producing cause-effect chains [20] and linking them to suspicious statements [3].

Harrold et al. [7] list nine classes of program

spectra, such as path count, data-dependency count, and execution trace. Among them, the execution trace spectrum is most widely used in debugging. Jones et al. [8, 10], in their work *Tarantula*, rank each statement according to suspiciousness, which is a function of the percentages of failed and successful test cases that execute the statement. Renieres and Reiss [17], in their work *NearestNeighbor*, find that the execution trace difference between a failed run and its nearest successful neighbor run is more effective for debugging. Baudry et al. [2] observe that some statements (known as a dynamic basic block) are always executed by the same set of test cases. They use a bacteriologic approach to generate test cases so as to maximize the number of dynamic basic blocks, and use the algorithm in [8, 10] to rank them. They further extend their work in [9] to make it possible for multiple developers to debug at the same time.

The most relevant related projects are CBI [12, 13] and SOBER [14, 15]. Rather than locating faulty statements, these techniques make use of predicates to indicate the faults. Since these techniques have been explained in Section 1, we do not repeat them here. Arumuga Nainar et al. [1] further extend CBI to address compound Boolean expressions. Zhang et al. [23] propose a fine-grained version of such techniques and use an empirical study to investigate the effectiveness.

6. Conclusion

Fault localization is a time-consuming and yet crucial activity in software debugging. Many previous studies contrast the feature spectra of successful and failed runs to locate the predicates correlated to faults (or locate the faulty statements directly). However, they overlook the investigation of the statistical distributions of the spectra, on which their parametric techniques fully rely. We have argued and empirically verified that assuming a specific distribution of feature spectra of dynamic program statistics is problematic. It highlights a threat to construct validity in fault localization research that previous studies do not report in their empirical evaluation and model development. We have also explained why parametric approximation is less desirable.

We have proposed a non-parametric approach that applies general hypothesis testing techniques proposed by mathematicians to statistical fault localization, and cast our technique in a predicate-based setting. We have conducted experiments on the Siemens suite to evaluate the effectiveness of our model. The experimental results show that our model can be effective in locating faults and requires no artificial parameters or operators. Empirically, our approach gives a better fault localization effectiveness than previous predicate-based fault localization techniques. Future work may include concurrent debugging of multi-fault programs. It will also be interesting to find out whether a non-parametric approach can be formally proven to be more suitable than a parametric approach.

References

[1] P. Arumuga Nainar, T. Chen, J. Rosin, and B. Liblit. Statistical debugging using compound Boolean predicates. In *Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2007*), pages 5–15. ACM Press, New York, NY, 2007.

[2] B. Baudry, F. Fleurey, and Y. Le Traon. Improving test suites for efficient fault localization. In *Proceedings of the 28th International Conference on Software Engineering* (*ICSE 2006*). ACM Press, New York, NY, 2006.

[3] H. Cleve and A. Zeller. Locating causes of program failures. In *Proceedings of the 27th International Conference on Software Engineering* (*ICSE 2005*), pages 342–351. ACM Press, New York, NY, 2005.

[4] J. S. Collofello and L. Cousins. Towards automatic software fault location through decision-to-decision path analysis. In *Proceedings of the 1987 National Computer Conference*, pages 539–544. Chicago, IL, 1987.

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

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

[7] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10 (3): 171–194, 2000.

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

[9] J. A. Jones, M. J. Harrold, and J. F. Bowring. Debugging in parallel. In *Proceedings of the 2007 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2007*), pages 16–26. ACM Press, New York, NY, 2007.

[10] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In

Proceedings of the 24th International Conference on Software Engineering (*ICSE 2002*), pages 467–477. ACM Press, New York, NY, 2002.

[11] B. Korel. PELAS: Program error-locating assistant system. *IEEE Transactions on Software Engineering*, 14 (9): 1253–1260, 1988.

[12] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI 2003), ACM SIGPLAN Notices*, 38 (5): 141–154, 2003.

[13] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI 2005), ACM SIGPLAN Notices*, 40 (6): 15–26, 2005.

[14] C. Liu and J. Han. Failure proximity: a fault localization-based approach. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 2006/FSE-14*). ACM Press, New York, NY, 2006.

[15] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: statistical model-based bug localization. In *Proceedings of the Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT International Symposium on Foundation of Software Engineering* (*ESEC 2005/FSE-13), ACM SIGSOFT Software Engineering Notes*, 30 (5): 286–295, 2005.

[16] R. Lowry. *Concepts and Applications of Inferential Statistics*. Vassar College, Poughkeepsie, NY, 2006. Available at http://faculty.vassar.edu/lowry/webtext.html.

[17] M. Renieres and S. P. Reiss. Fault localization with nearest neighbor queries. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering* (*ASE 2003*), pages 30–39. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[18] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3 (3): 121–189, 1995.

[19] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10 (4): 352–357, 1984.

[20] A. Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 2002/FSE-10), ACM SIGSOFT Software Engineering Notes*, 27 (6): 1–10, 2002.

[21] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28 (2): 183–200, 2002.

[22] X. Zhang, N. Gupta, and R. Gupta. Locating faults through automated predicate switching. In *Proceedings of the 28th International Conference on Software Engineering* (*ICSE 2006*), pages 272–281. ACM Press, New York, NY, 2006.

[23] Z. Zhang, B. Jiang, W. K. Chan, and T. H. Tse. Debugging through evaluation sequences: a controlled experimental study. In *Proceedings of the 32nd Annual International Computer Software and Applications Conference* (*COMPSAC 2008*), volume 1. IEEE Computer Society Press, Los Alamitos, CA, 2008.