Fault localization through evaluation sequences ☆

Zhenyu Zhang [a], Bo Jiang [a], W.K. Chan [b], T.H. Tse [a,*], Xinming Wang [c]

[a] Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong
[b] Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong
[c] Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Kowloon, Hong Kong

ARTICLE INFO

ABSTRACT

Predicate-based statistical fault-localization techniques find fault-relevant predicates in a program by contrasting the statistics of the evaluation results of individual predicates between failed runs and successful runs. While short-circuit evaluations may occur in program executions, treating predicates as atomic units ignores this fact, masking out various types of useful statistics on dynamic program behavior. In this paper, we differentiate the short-circuit evaluations of individual predicates on individual program statements, producing one set of evaluation sequences per predicate. We then investigate experimentally the effectiveness of using these sequences to locate faults by comparing existing predicate-based techniques with and without such differentiation. We use both the Siemens program suite and four real-life UNIX utility programs as our subjects. The experimental results show that the proposed use of short-circuit evaluations can, on average, improve predicate-based statistical fault-localization techniques while incurring relatively small performance overhead.

© 2009 Elsevier Inc. All rights reserved.

1. Introduction

Software debugging is a crucial development activity, which may easily take up a significant amount of resources in a typical software development project. It has at least three major tasks, namely, fault localization, fault repair, and regression testing of the repaired program. Among them, fault localization has been recognized as the hardest, most tedious, and most time-consuming (Vessey, 1985). Using an effective fault-localization technique to assist programmers to find bugs is a long-standing trend to alleviate the problem.

Many kinds of fault-localization technique have been proposed. One of them is to apply a statistical approach to correlating program entities (such as statements) with failures. A key insight is based on the assumption that certain dynamic *feature* of program entities is more sensitive to the *difference* between the set of failed runs and the set of all (or successful) runs. Thus, there are two key elements underlying the successful applications of such class of dynamic analysis techniques: First, a technique should use a feature (or a set of features) to measure sensitivity. Second, the tech-

nique should have a function to compare sensitivity values. The function essentially ranks the sensitivity values in a total order. For example, techniques such as Liblit et al. (2005) and Liu et al. (2006) produce a real number value to represent sensitivity, and sort these values in ascending or descending order. The relative magnitudes of sensitivity values (rather than their absolute values, since the value ranges can be unbounded in general) are popularly used when ranking the program entities. By mapping the relative order of the sensitivity values back to the associated program entities, the techniques can produce a ranked list of program entities accordingly.

One strategy (Chilimbi et al., 2009; Liblit et al., 2005; Liu et al., 2006) is to use predicates as program entities, and the execution counts and execution results as the dynamic features. They are used to estimate the difference between the probability of failed runs and that of successful runs, and then sort the program entities accordingly. By sampling selected predicates, rather than all predicates or statements, this strategy reduces the overhead in collecting debugging information. It also avoids disclosing a particular aspect of every execution, such as which statements have been executed. Hence, it lowers the risk of information leakage, which is a security concern.

Such a strategy, however, needs to summarize the execution statistics of individual predicates. A compound predicate may be executed in one way or another owing to short-circuit evaluations over different sub-terms of the predicate. The execution statistics of a compound predicate is, therefore, the summary of a collection

---

of lower-tier evaluations over different sub-terms (Zhang et al., 2008). *Is differentiating such lower-tier evaluations beneficial in improving the effectiveness of predicate-based fault-localization techniques?* This paper conducts a controlled experiment to investigate the impact of the use of short-circuit evaluation sequences to improve statistical fault-localization techniques.

We first give a few preliminaries. A *passed test case* is one that shows no failure, and a *failure-causing test case* is one identified to have detected a failure. A typical program contains numerous *predicates* in branch statements such as if- and while-statements. (Some programming languages like C further allow predicates on assignment statements.) These predicates are in the form of *Boolean expressions*, such as "`*j <= l || src[*i+l] == '\0'`", which may comprise further *conditions*, such as "`*j <= l`" and "`src[*i+l] == '\0'`".

Previous studies on statistical fault localization (Chilimbi et al., 2009; Liblit et al., 2005; Liu et al., 2006) find the fault-relevant predicates in a program by counting the number of times (n_t) a predicate is evaluated to be `true` in an execution as well as the number of times (n_f) it is evaluated to be `false`, and then comparing these counts in various ways. The *evaluation bias* $\frac{n_t}{n_t+n_f}$ of a predicate (Liu et al., 2006) is the percentage that it is evaluated to be `true` among all evaluations in a run.

The SOBER approach (Liu et al., 2006) proposes to contrast the differences between a set of evaluation biases due to passed test cases and that due to failure-causing ones for every predicate in the program. It hypothesizes that, the greater is the difference between such a pair of sets of evaluation biases, the higher will be the chance that the corresponding predicate is fault-relevant. The CBI approach (Liblit et al., 2005) proposes a heuristic that measures the increase in probability that a predicate is evaluated to be `true` in a set of failure-causing test cases, compared to the whole set of (passed and failure-causing) test cases. These proposals are particularly interested in the evaluation results of predicates. They use the resultant values of the predicates to determine the ranks.

A predicate can be semantically modeled as a Boolean expression. As mentioned above and to be discussed in Section 2, the resultant values of a Boolean expression may be calculated from different evaluation sequences or from the whole predicate as one unit. If we ignore the information on evaluation sequences, we may be masking out useful statistics for effective fault localization. In this paper, we investigate whether the effect of a lower-tier concept—evaluation sequences – of predicates can be a significant factor affecting the effectiveness of predicate-based statistical fault localization. We set up a controlled experiment to study this question. We further investigate the performance overhead of our technique using programs of different scales, analyze the complexity of our technique, and report empirical results regarding the time-cost of applying our technique.

The major contribution of this paper is twofold: (i) We provide the first set of experimental results regarding the effect of short-circuit evaluations on statistical fault localization. (ii) We show that short-circuit evaluation has a significant impact on the effectiveness of predicate-based fault-localization techniques.

We will illustrate the potential of using evaluation sequences for fine-grained statistical fault localization in Section 2, which casts a scene for us to formulate the research questions in Section 3, followed by the associated experiment in Section 4. We will then review related work in Section 5. Section 6 concludes the paper.

2. A motivating study

This section shows a motivating study we have conducted. It enables readers to have a feeling of how the distribution of evaluation biases at the evaluation sequence level can be used to pin-point a fault-relevant predicate (which also happens to be a faulty predicate in this example).

The upper part of Fig. 1 shows a code fragment excerpted from the original version (version v0) of `print_tokens2` from the Siemens suite of programs (Do et al., 2005). We have labeled the three individual conditions as C_1, C_2, and C_3, respectively. The lower part of the same figure shows the code fragment excerpted from a faulty version (version v8) of the Siemens suite, where a fault has been seeded into the predicate by adding an extra condition "`ch == '\t'`". We have labeled this condition as C_4.

Because of the effect of short-circuit rules of the C language on Boolean expressions, a condition in a Boolean expression may be evaluated to be `true` or `false`, or may not be evaluated at all (denoted by $\bot$). Furthermore, in terms of evaluations, the conditions on a Boolean expression can be seen as an ordered sequence. In most cases, when a preceding condition in an evaluation sequence is not evaluated, by the short-circuit rule, no succeeding condition in the evaluation sequence will be evaluated.

For the faulty Boolean expression in the fragment shown in Fig. 1, there are five legitimate evaluation sequences es_1 to es_5, as shown in Table 1. The columns under the individual conditions C_1 to C_4 represent the evaluation outcomes of the respective conditions based on the short-circuit rules of the programming language. In the column entitled v0, it shows the respective resultant values of the predicate in the original version of the program. In this column, the last two grids are merged because the two evaluation sequences (es_4 and es_5) make no difference in the original program. The column entitled v8 shows the respective resultant values in the faulty program. The rightmost column shows whether the original and faulty predicates give the same values.

To gain an idea of whether short-circuit rules can be useful for fault localization, we have run an initial experiment. We apply the whole test pool for the program from the Software-artifact Infrastructure Repository (SIR) (Do et al., 2005), and record the counts of each of the five evaluation sequences for each test case. Following Liu et al. (2006), we use the formula $\frac{n_t}{n_t+n_f}$ in Section 1 to calculate the evaluation biases for the set of passed test cases, and those for the set of failure-causing test cases. The results are shown as the histograms in Figs. 2 and 3. The distribution of evaluation

```
'* Original Version v0 */
        C1              C2              C3
.f (ch == ' ' || ch == '\n' || ch == 59)
    return(true);


'* Faulty Version v8 */
        C1              C2              C3              C4
.f (ch == ' ' || ch == '\n' || ch == 59 || ch == '\t'
    return(true);
```

Fig. 1. Code excerpts from versions v0 and v8 of `print_tokens`.

Table 1
Evaluation sequences of code fragments.

Evaluation sequence	C_1	C_2	C_3	C_4	v0	v8	v0 = v8?
es_1	true	$\bot$	$\bot$	$\bot$	true	true	yes
es_2	false	true	$\bot$	$\bot$	true	true	yes
es_3	false	false	true	$\bot$	true	true	yes
es_4	false	false	false	true	false	true	**no**
es_5	false	false	false	false		false	yes

(a) Results of evaluation sequence es_1

(b) Results of evaluation sequence es_2

(c) Results of evaluation sequence es_3
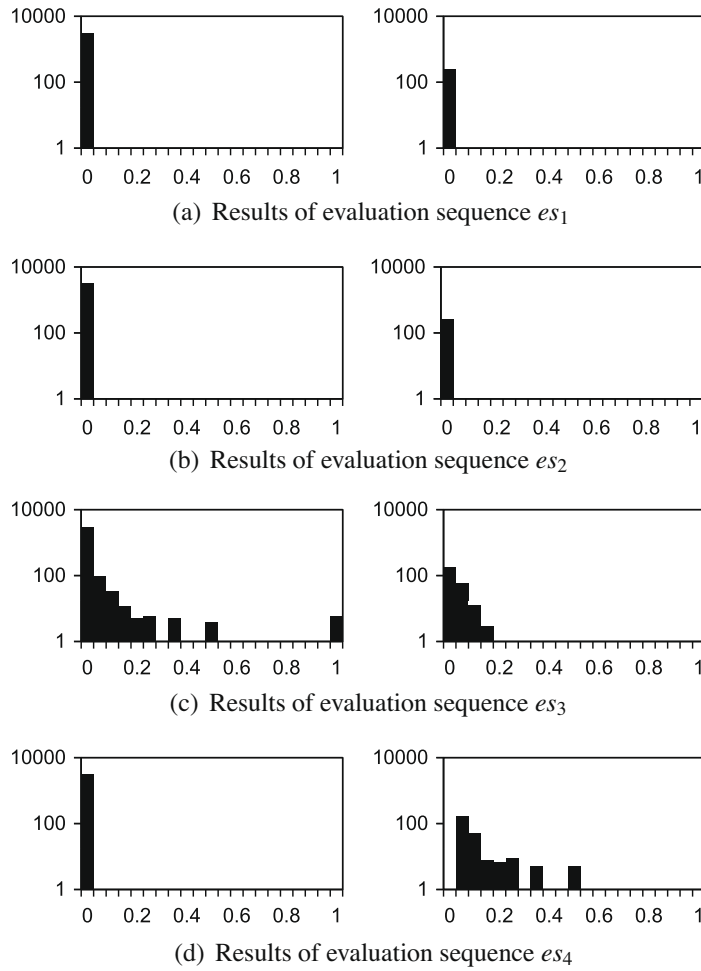

(d) Results of evaluation sequence es_4

Fig. 2. Comparisons of distributions of evaluation biases for evaluation sequences es_1, es_2, es_3, and es_4 (x-axis: evaluation bias; y-axis: number of test cases).
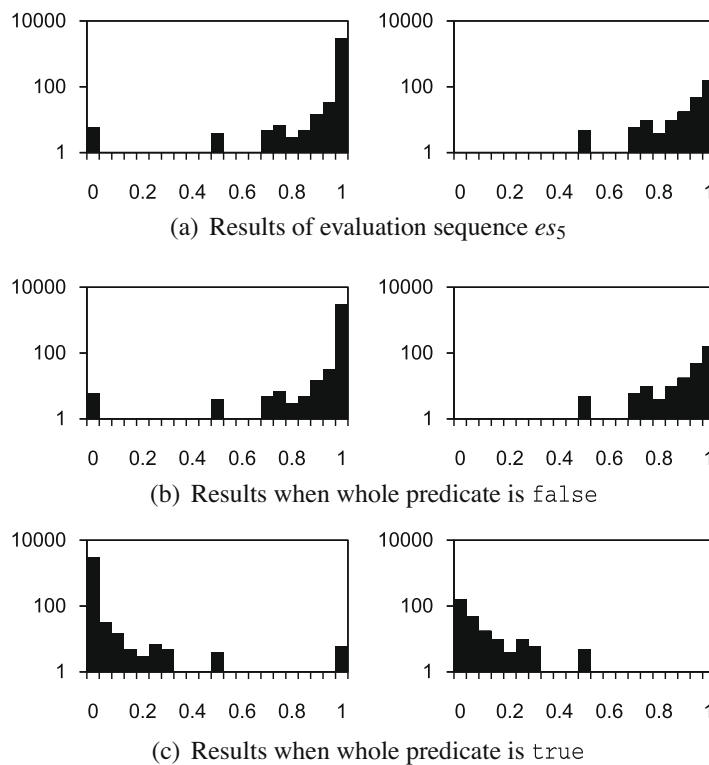

(a) Results of evaluation sequence es_5

(b) Results when whole predicate is `false`

(c) Results when whole predicate is `true`

Fig. 3. Comparisons of distributions of evaluation biases for evaluation sequences es_5 and the whole predicate (x-axis: evaluation bias; y-axis: number of test cases).

biases over passed test cases and that over failure-causing test cases are shown side by side for comparison. Figs. 2a–d and 3a show the comparative distributions of the five evaluation sequences. Figs. 3b and 3c are the comparative distributions for the whole predicate (when evaluated to be `true` and when evaluated to be `false`, respectively), as used in Liu et al. (2006).

From the histograms in Figs. 2 and 3, we observe that the distribution of evaluation biases for es_4 on passed test cases is drastically different from that of the failure-causing ones. Indeed, it is the most different one among all pairs of histograms shown in the figures. We also observe from Table 1 that the fault in the code fragment can only be revealed when es_4 is used, and the fault does not affect the values in the other alternatives.

Our initial study indicates that it may be feasible to use evaluation sequences to identify a faulty predicate more accurately. When a fault is not on the predicate, most predicate-based techniques facilitate fault localization by ranking the predicates in order of their relations to the fault. In such a case, our technique also works by finding the fault-relevant predicates rather than the actual faulty statement. However, it is still uncertain how much the use of evaluation sequences will be beneficial to fault localization. We will formulate our research questions in the next section and then investigate them experimentally in Section 4.

3. Research questions

In this section, we will discuss the research questions to be addressed by our controlled experimental study. We refer to a predicate-based statistical fault-localization technique as a *base* technique, and refer to the use of evaluation sequences in predicate execution counts as the *fine-grained* version of the base technique.

RQ1: In relation to the base technique, is the use of evaluation sequences for statistical fault localization effective?

RQ2: If the answer to **RQ1** is true, is the effectiveness of using evaluation sequences significantly better than the base technique?

RQ3: Do the execution statistics of different evaluation sequences of the same predicate differ significantly?

3.1. Performance evaluation

Performance metrics are widely used to facilitate comparisons among different approaches. Renieris and Reiss (2003) propose a metric (T-score) for measuring their fault-localization technique. The method is also adopted by Cleve and Zeller (2005) and Liu et al. (2006) to evaluate other fault-localization techniques.

For ease of comparison with previous work, we also use the T-score metric to evaluate the fine-grained evaluation sequence approach in relation to the corresponding base techniques. We select two base techniques for study, namely SOBER (Liu et al., 2006) and CBI (Liblit et al., 2005), because they are representative predicate-based fault-localization techniques. Both of them take Boolean expressions in conditional statements and loops as predicates and generate a ranked list showing, in descending order, how much each of these predicates is estimated to be related to a fault. Both techniques have been evaluated in a previous study (Liu et al., 2005) using the T-score metric (Renieris and Reiss, 2003) and compared with other fault-localization techniques. Follow-up studies such as Arumuga Nainar et al. (2007) and Chilimbi et al. (2009) have been derived from these techniques. Since the follow-up work is based on the same framework and hence similar in nature, we will only use the base versions to investigate whether we may improve on them using our approach. If our approach may indeed improve on the two base techniques, their derived versions should also benefit from it.

In brief, the T-score metric takes a program P, its marked faulty statements S, and a sequence of most suspected faulty statements S' as inputs, and produces a value V as output. The procedure to compute the T-score is as follows: (i) Generate a Program Dependence Graph (PDG) G for P. (ii) Using the dependence relations in the PDG as a measure of distance among statements, do a breadth-first search starting with the statements in S', until some statement in S is reached. (iii) Return the percentage of searched statements (with respect to the total number of statements in P) as the value V. If the original S' consists of k most suspicious faulty statements, the final result is known as the top-k T-scores.

This measure is useful in assessing objectively the quality of proposed ranking lists of fault-relevant predicates and the performance of fault-localization techniques. Since the evaluation sequence approach is built on top of base techniques (such as SOBER and CBI), we also use T-scores to compare different approaches in our controlled experiment to answer the research questions.

3.2. Enabling fine-grained view of base techniques

As we are interested in studying the impact of short-circuit evaluations and evaluation sequences for statistical fault localization, we need a method to incorporate the fine-grained view into a base technique. Intuitively, this will provide execution statistics that may help statistical fault-localization techniques identify the locations of faults more accurately.

We note that a base technique, such as SOBER or CBI, conducts sampling of the predicates in a subject program to collect run-time execution statistics, and ranks the fault relevance of the predicates. To assess the effectiveness of the selected set of predicates to locate faults, researchers may use the T-score metric to determine the percentage of code examined in order to discover the fault. As such, given a set of predicates applicable to a base technique, we identify all the potential evaluation sequences for each of the predicates. We then insert probes at the predicate locations to collect the evaluation outcomes of atomic conditions in these predicates. Based on the evaluation outcomes of the atomic conditions, we can determine the evaluation sequence that takes place for every predicate. For each individual evaluation sequence, we count the number of times it is exercised with respect to every test case. By treating each evaluation sequence as a distinct (fine-grained) predicate in the base technique, the ranking approach in the base technique can be adopted to rank these fine-grained predicates.

On the other hand, from the developers' viewpoint, it may be more convenient to recognize (through their eyeballs) the occurrence of an original predicate from the code, rather than an evaluation sequence of the predicate. Hence, it is to the benefit of developers to map the ranked evaluation sequences to their respective predicates, and thus to the corresponding statements.

Some measures need to be taken in the above mapping procedure. Different evaluation sequences may receive different ranks. A simple mapping may thus result in a situation where a predicate occurs more than once in a ranking list. We choose to use the highest rank of all evaluation sequences for each individual predicate as the final rank of that predicate. This strategy also aligns with the basic idea of predicate ranking in SOBER and CBI. We refer to the fine-grained approach as *Debugging through Evaluation Sequences* (*DES*). Let us take the motivating example in Section 2 as an illustration. In previous predicate-based approaches such as SOBER (Liu et al., 2005), the Boolean expression "ch == '' ‖ ch == '\n' ‖ ch == 59 ‖ ch == '\t'" is used as one predicate. When the Boolean expression is `true` or `false`, previous techniques

evaluate the predicate as `true` or `false`, respectively, and records its evaluation biases accordingly. In our approach, we form a fine-grained viewpoint and investigate four atomic Boolean expressions "`ch == ' '`", "`ch == ' \n'`", "`ch == 59 `", and "`ch == ' \t'`" as shown in Table 1. The evaluation sequence es_2, for instance, shows the case where "`ch == ' '`" is evaluated to be `false`, "`ch == ' \n'`" is evaluated to be `true`, and the evaluations of the other two atomic Boolean expressions "`ch == 59 `" and "`ch == ' \t'`" are short-circuited. Each time the Boolean expression is evaluated, it must fall into one (and only one) of the evaluation sequences es_1 to es_5. We regard the "falling into" or "not falling into" each evaluation sequence by the Boolean expression as a kind of predicate for that evaluation sequence. For example, if the evaluation of the Boolean expression falls into the evaluation sequence es_2, we regard the corresponding predicate with respect to es_2 as evaluated to be `true`; and if the evaluation of the Boolean expression falls into another evaluation sequence, we regard the corresponding predicate with respect to es_2 as evaluated to be `false`. We record the evaluation biases for this kind of predicate accordingly, and adapt previous techniques to work on the evaluation sequence level.

4. Controlled experiment

This section presents a controlled experiment and its results and analyses.

4.1. Subject programs and test cases

In this study, we choose the Siemens suite of programs as well as four UNIX utility programs to conduct our experiment.

The Siemens programs were originally created to support research on data-flow and control-flow test adequacy (Hutchins et al., 1994). Our version of the Siemens programs is obtained from the Software-artifact Infrastructure Repository (SIR) (Do et al., 2005) at http://sir.unl.edu. The Siemens suite consists of seven programs as shown in Table 2. A number of faulty versions are attached to each program. In our experiment, if any faulty version comes with no failure-causing cases, we do not include it in the experiment, since the base techniques (Liblit et al., 2005; Liu et al., 2006) require failure-causing test cases. We use a UNIX tool, gcov, to collect the execution statistics needed for computation. Six faulty versions that cannot be processed by gcov are excluded. As a result, we use 126 faulty versions in total.

Since the Siemens programs are of small sizes and the faults are seeded manually, we also use medium-sized real-life UNIX utility

programs with real and seeded faults as additional subjects to strengthen the external validity of our experiment. These programs, also from SIR, include flex, grep, gzip, and sed as shown in Table 2. Each of these programs has one or more versions and each version contains dozens of single faults. We create one faulty program for each single fault, apply the same strategy above to exclude problematic ones, and use a total of 110 faulty versions as target programs.

Each of the Siemens and UNIX programs is equipped with a test pool. According to the authors' original intention, the test pool simulates a representative subset of the input domain of the program, so that test suites should be drawn from such a test pool (Do et al., 2005). In the experiment, we follow the work of Liu et al. (2006) to input the whole test pool to every technique to rank predicates or their evaluation sequences.

Table 2 shows the statistics of the subject programs and test pools that we use. The data with respect to each subject program, including the executable lines of code (column "LOC"), the number of faulty versions (column "# of Versions"), the size of the test pool (column "# of Cases"), the number of Boolean expressions (column "# of Bools"), the average percentage of Boolean expression statements with respect to all statements (column "% of Boo"), and the average percentage of compound Boolean expression statements with respect to all Boolean expression statements (column "% of Com"), are obtained from SIR (Do et al., 2005) as at January 10, 2008. Since the subject programs print_tokens and print_tokens2 have similar structures and functionality, and each has only a few faulty versions (which cannot give meaningful statistics), we show their combined results in the figure. (By the same token, the combined results of schedule and schedule2 are shown in Fig. 7.) For instance, there are 10 faulty versions for the print_tokens2 program. Their sizes vary from 350 to 354 LoC, and their test pool contains 4115 test cases. On average, 5.4% of the Boolean expression statements in these faulty versions contain compound Boolean expressions. Other rows can be interpreted similarly. We note that many faults in these faulty versions do not occur in predicates.

We observe from the column "% of Com" that, in each subject program, the percentage of predicates having more than one atomic condition is low. This makes the research questions even more interesting: We would like to see whether such a low percentage would affect the performance of a base technique to a large extent.

4.2. Setup of controlled experiment

In this section, we describe the setup of the controlled experiment. Using our tool, we produce a set of instrumented versions

Table 2
Statistics of subject programs.

Program	LOC	# of Versions	# of Cases	# of Bools	% of Bools	% of Compounds
print_tokens (2 programs)	341–354	17	4130	81	23.7	1.7
replace	508–515	31	5542	66	12.9	2.0
schedule (2 programs)	261–294	14	2710	43	16.4	1.0
tcas	133–137	41	1608	11	8.1	2.4
tot_info	272–274	23	1052	46	16.8	5.6
Average	310	18	3115	55	16.9	3.0
flex (2.4.7–2.5.4)	8571–10,124	21	567	969	10.3	5.5
grep (2.2–2.4.2)	8053–9089	17	809	930	10.9	14.1
gzip (1.1.2–1.3)	4081–5159	55	217	591	12.7	11.6
sed (1.18–3.02)	4756–9289	17	370	552	7.8	11.6
Average	7390	28	491	761	10.4	10.7

Legion: LOC: executable lines of code.
# of Versions: number of faulty versions.
# of Cases: number of test cases in the test pool.
# of Bools: average number of Boolean expressions.
% of Bools: average percentage of Boolean expression statements with respect to all statements.
% of Compounds: average percentage of compound Boolean expressions with respect to all Boolean expressions.

of the subject programs, including both the original and faulty versions. Based on the instrumentation log as well as the coverage files created by gcov, we calculate the execution counts for the evaluation sequences, and finally rank the Boolean expression statements according to the description presented in Section 3. We also calculate the number of faults successfully identified through the examined percentage of code at different T-scores (see Section 3).

The experiment is carried out on a DELL PowerEdge 1950 server with two 4-core Xeon 5355 (2.66 Hz) processors, 8GB physical memory and 400 GB hard disk equipped, serving Solaris UNIX with the kernel version of Generic_120012-14.

Our experimental platform is constructed using the tools of flex++ 2.5.31, bison++ 1.21.9-1, CC 5.8, bash 3.00.16(1)-release (i386-pc-solaris2.10), and sloccount 2.26.

4.3. Results and analysis

In this section, we present the experimental results, compare the relative improvements in effectiveness of the integrated approach with respect to the base techniques, and address the research questions one by one.

4.3.1. Overall results of DES-enabled techniques

Fig. 4a compares the results by SOBER and DES-enabled SOBER on all 11 programs, and Fig. 4b compares those by CBI and DES-enabled CBI on the same programs. For ease of discussion, we refer to DES-enabled SOBER as DES_SOBER, and DES-enabled CBI as DES_CBI.

The x-axis of each plot in these two figures shows the T-scores, each of which represents the percentage of statements of the respective faulty program version to be examined. The y-axis is the percentage of faults located within the given code-examination range. According to Liu et al. (2006), the use of the top 5 predicates in the ranked list will produce the best results for both SOBER and CBI. For a fair comparison with previous work, we also adopt the use of the top 5 predicates in the controlled experiment. In the remaining parts of the paper, therefore, we will always compare the top-5 T-scores for DES_SOBER and DES_CBI against those for SOBER and CBI.

We observe from Fig. 4a that DES_SOBER consistently achieves better average fault-localization results (that is, more faults for the same percentage of examined code) than SOBER. For example, when examining 10% of the code, DES_SOBER can find about 5% more faults than SOBER. As the percentage of examined code increases, however, the difference shrinks. This is understandable because, when an increasing amount of code has been examined, the difference between marginal increases of located faults will naturally be diminished. When all the faults are located or all the statements are examined, the two curves will attain the same percentage of located faults. We also observe from Fig. 4b that DES_CBI also outperforms CBI. When examining 10% of the code, DES_CBI can find about 10% more faults than CBI.

(a) SOBER on all programs

(b) CBI on all programs
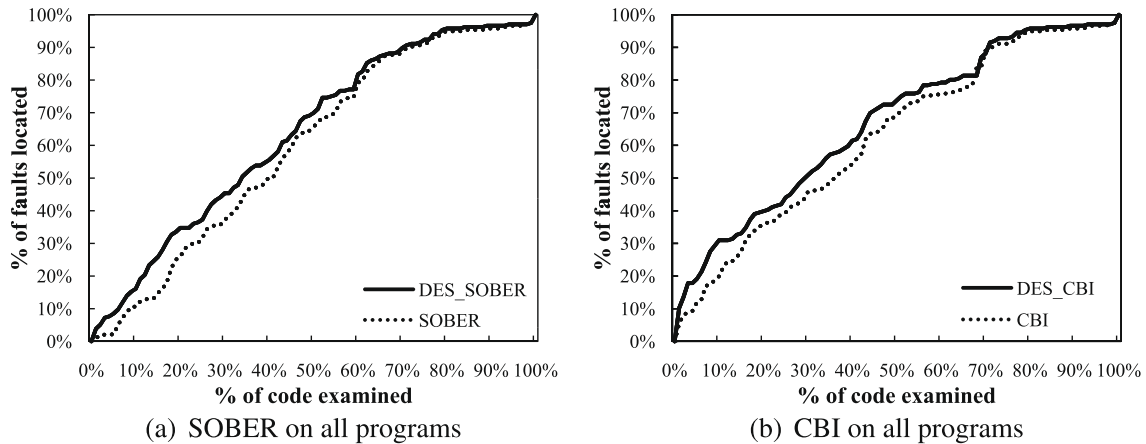
Fig. 4. Comparisons of DES-enabled techniques with base techniques on all programs.

(a) SOBER on print_tokens and print_tokens2 programs

(b) CBI on print_tokens and print_tokens2 programs
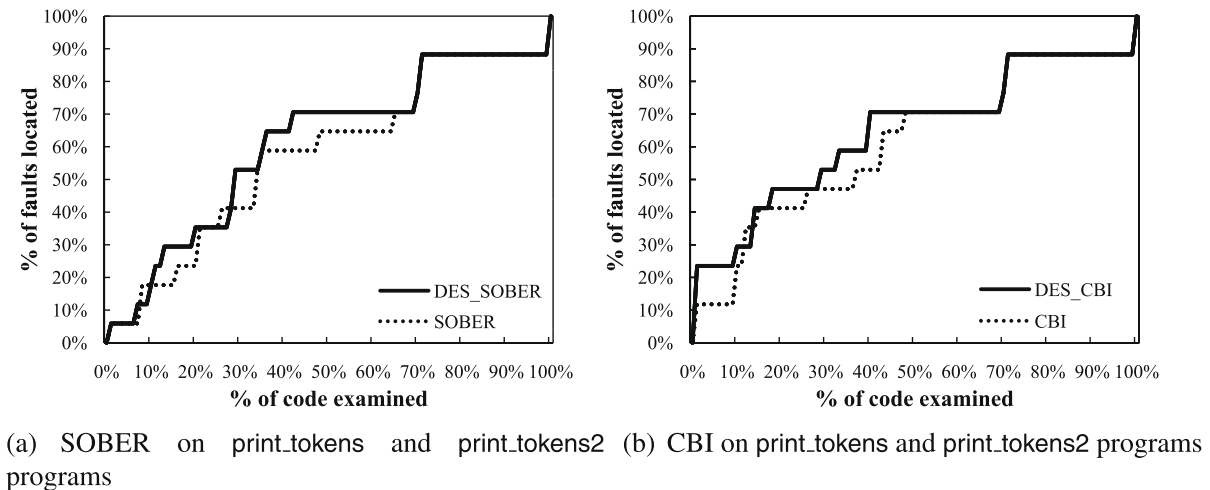
Fig. 5. Comparisons of DES-enabled techniques with base techniques on print_tokens and print_tokens2 programs.

4.3.2. Individual results of DES-enabled techniques

To further verify whether the above results generally hold for all the programs, we examine the outcomes of each individual program, as shown in Figs. 5–13.

Let us first focus on Fig. 5. It shows the results of CBI, DES-CBI, SOBER, and DES_SOBER on the print_tokens and print_tokens2 programs. For these two programs, DES_SOBER outperforms SO-

BER for almost the entire code-examination range from 0 to 100%, except two short ranges around 10 and 30%. Similarly, DES_CBI performs better than CBI almost throughout the range from 0 to 100%.

Let us move on to the replace program. DES_SOBER and DES_CBI again exhibit advantage over SOBER and CBI, respectively, almost throughout the entire range from 0 to 100%.
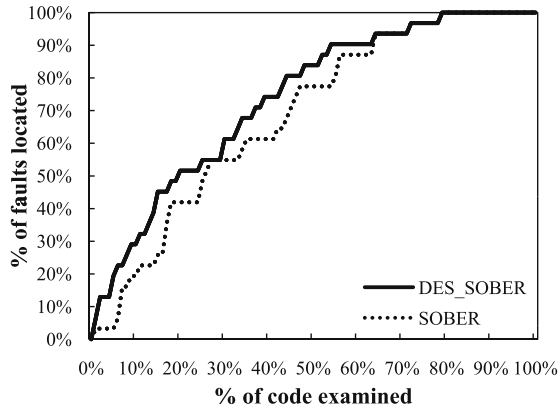
(a) SOBER on replace program   (b) CBI on replace program
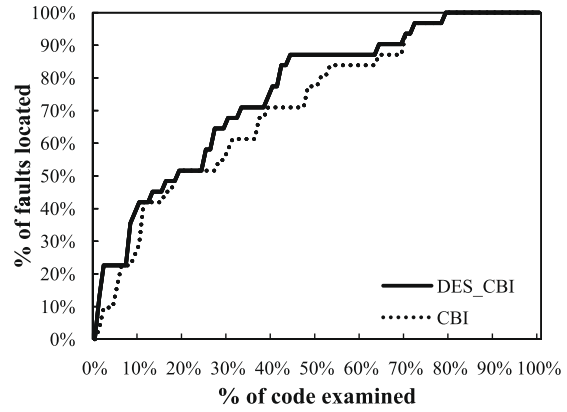
Fig. 6. Comparisons of DES-enabled techniques with base techniques on replace program.
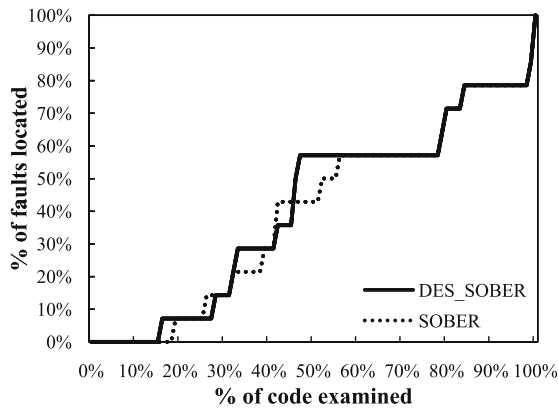
(a) SOBER on schedule and schedule2 programs   (b) CBI on schedule and schedule2 programs
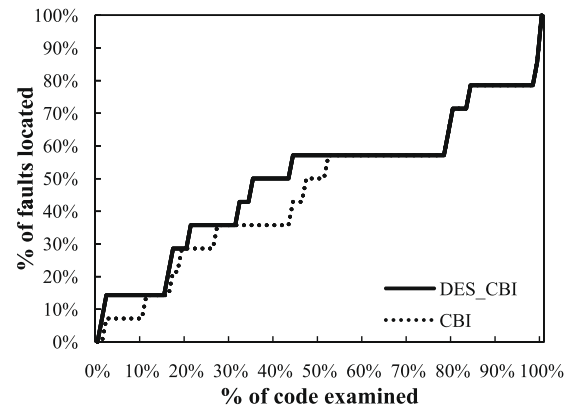
Fig. 7. Comparisons of DES-enabled techniques with base techniques on schedule and schedule2 programs.
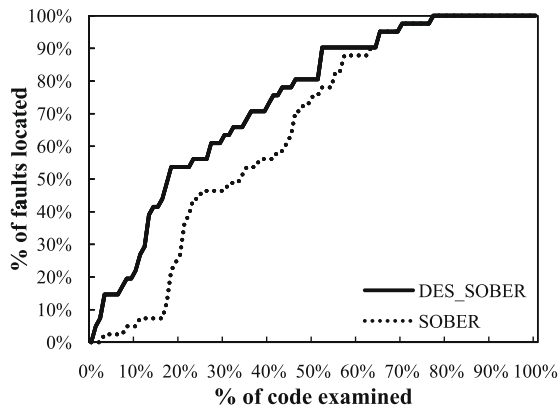
(a) SOBER on tcas program   (b) CBI on tcas program
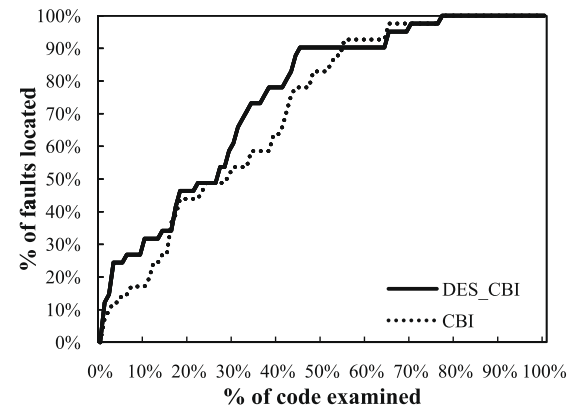
Fig. 8. Comparisons of DES-enabled techniques with base techniques on tcas program.

(a) SOBER on tot_info program

(b) CBI on tot_info program
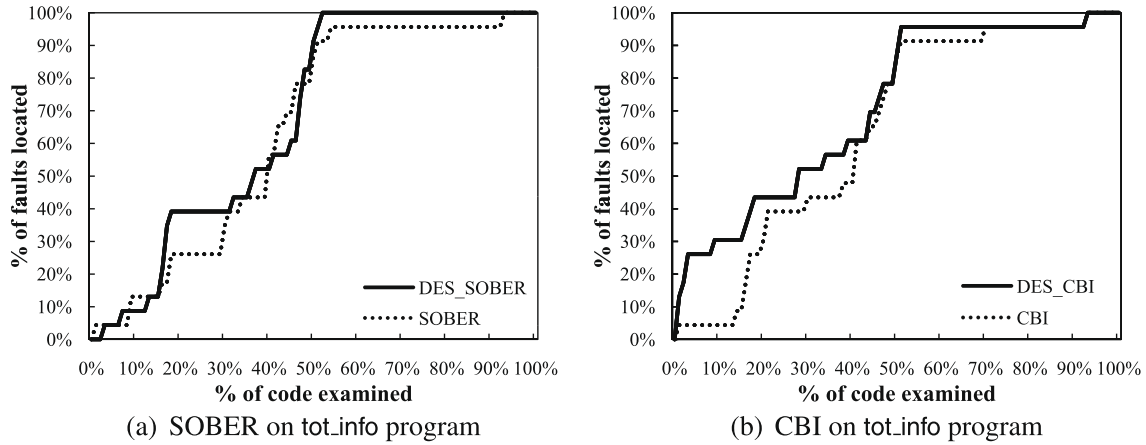
Fig. 9. Comparisons of DES-enabled techniques with base techniques on tot_info program.

(a) SOBER on flex program

(b) CBI on flex program
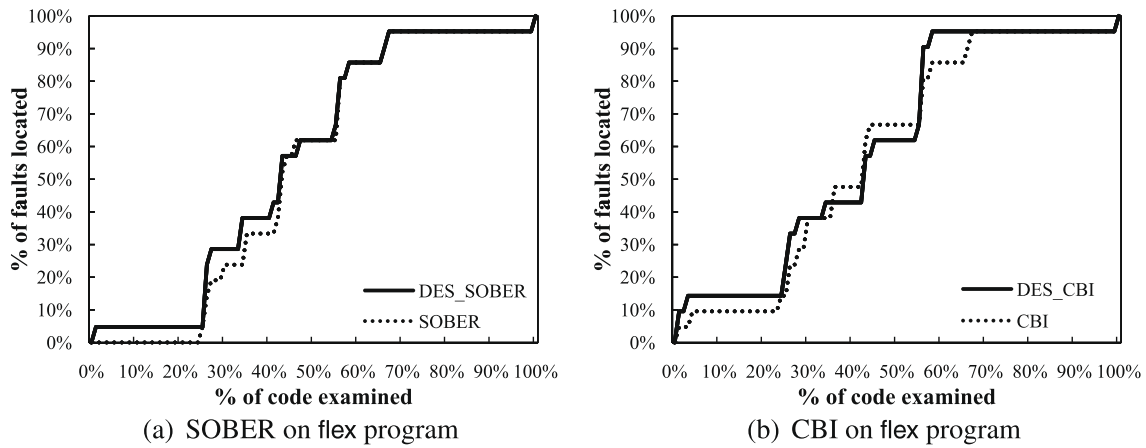
Fig. 10. Comparisons of DES-enabled techniques with base techniques on flex program.

(a) SOBER on grep program

(b) CBI on grep program
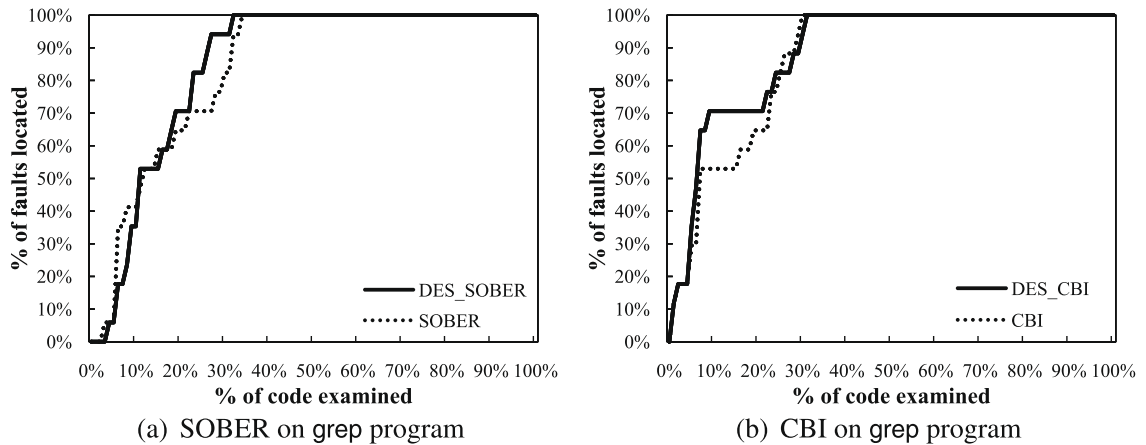
Fig. 11. Comparisons of DES-enabled techniques with base techniques on grep program.

For the programs schedule and schedule2, neither DES_SOBER nor SOBER shows advantage over each other. However, for the same programs, DES_CBI shows advantage over CBI throughout the range from 0 to 100%.

For the tcas program, DES_SOBER and DES_CBI obviously perform better than SOBER and CBI, respectively, except that DES_CBI is caught up by CBI when examining more than 60% code.

For the tot_info program, DES_SOBER shows great advantage over SOBER in the code-examination range from 20 to 30%.

DES_SOBER also shows continuous and steady advantage over SOBER in the range from 50 to 90%. In the remaining ranges, DES_SOBER and SOBER perform comparably. At the same time, DES_CBI shows observable advantage over CBI.

We next move to the flex program. DES_SOBER outperforms SOBER in the code-examination range of 0–45%, after which they show comparable effectiveness. However, for the same program flex, neither CBI nor DES_CBI shows consistent advantage over each other. CBI is more effective than DES_CBI in the code-examination

(a) SOBER on gzip program

(b) CBI on gzip program
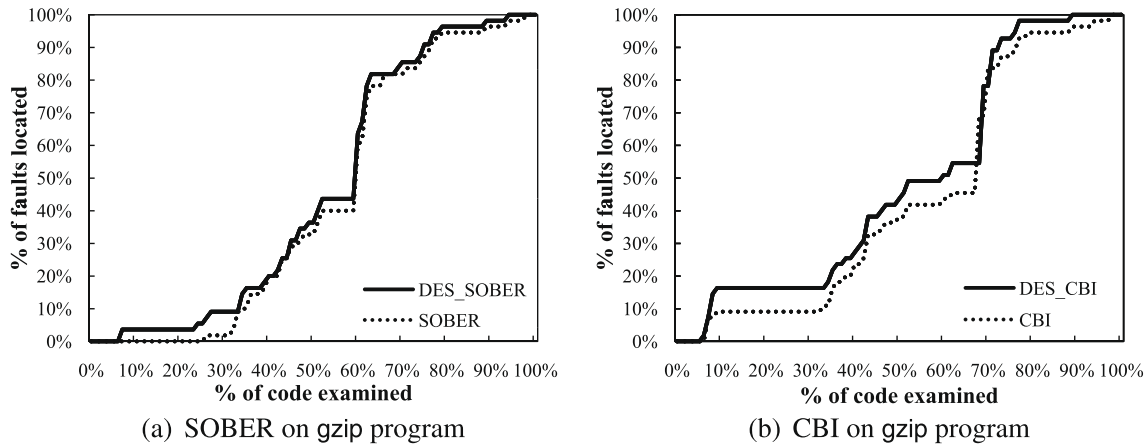
Fig. 12. Comparisons of DES-enabled techniques with base techniques on gzip program.

(a) SOBER on sed program
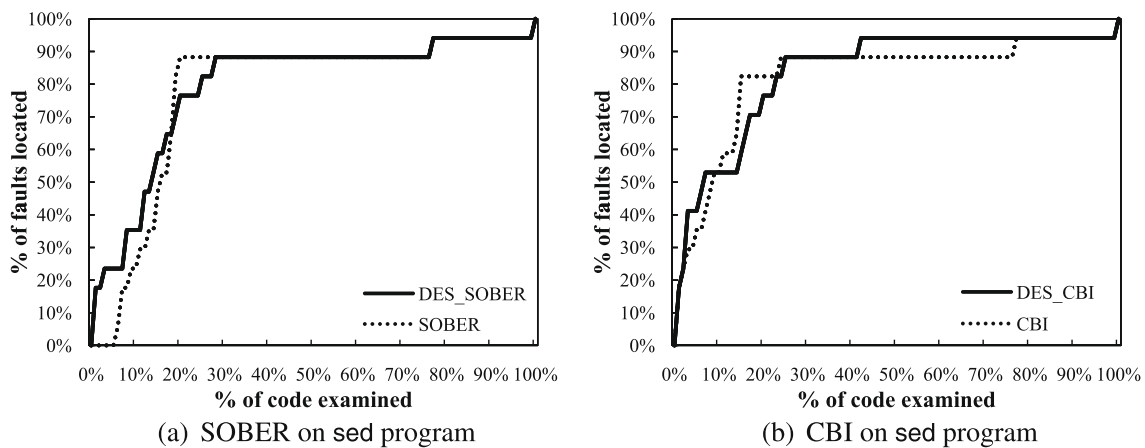
(b) CBI on sed program

Fig. 13. Comparisons of DES-enabled techniques with base techniques on sed program.

range of about 35–55%. In other ranges, DES_CBI is more effective than CBI. On average, they perform comparably to each other.

For the grep program, DES_CBI noticeably outperforms CBI, while there is no obvious difference between DES_SOBER and SOBER. In the first 10% code-examination range, SOBER locates more faults than DES_SOBER, but is caught up when examining 10–20% of the code. For both DES_SOBER and SOBER, all the faults in the faulty versions of grep can be located when examining up to 40% of the code. In short, their effectiveness is also comparable.

For the gzip program, both DES_SOBER and DES_CBI locate more faults than SOBER and CBI, respectively, in the entire code-examination range.

The comparison results for the sed program are like those for the flex program. DES_SOBER shows an observable advantage over SOBER, while neither DES_CBI nor CBI shows steady advantage over each other. CBI catches up with DES_CBI only in the code-examination range of 10–30%. DES_CBI always locates more faults than CBI in other code-examination ranges.

In summary, we observe that, on average, the DES-enabled techniques are comparable to, if not more effective than, their respective base techniques for the programs under study.

4.3.3. Answering RQ1: Is DES effective?

From the results of the Siemens suite of programs (Figs. 5–9), we have just observed that the DES-enabled techniques are at least comparable to their base counterparts. However, the Siemens programs are small in size. To further generalize our findings, we have also studied the DES approach on four UNIX utility programs. The results are similar. Since both SOBER and CBI are deemed as effective techniques in previous studies (Liu et al., 2005), we can, therefore, answer the first research question—the DES approach *is* effective.

4.3.4. Answering RQ2: Is DES better?

Our intuitive observation above, drawn from Figs. 5–13, is that the DES approach is comparable to, if not more effective than, the respective base fault-localization technique. We are interested in finding out whether, on average, there is significant advantage in using the DES-enhanced fault-localization techniques over the use of the base techniques.

To do that, we compare for each program the relative improvements in effectiveness of the DES-enabled versions with respect to the base techniques, as shown in Table 3. For each program having n faulty versions, we use $C_i, DC_i, S_i,$ and DS_i to represent the T-scores of CBI, DES_CBI, SOBER, and DES_SOBER for the i-th faulty version. We compute $(C_i - DC_i)/C_i$ and $(S_i - DS_i)/S_i$ to estimate the relative improvements in effectivensss when the respective techniques are DES-enabled. We then calculate the mean and standard deviation for the full set of these values (that is, $\{(C_1 - DC_1)/C_1, (S_1 - DS_1)/S_1, (C_2 - DC_2)/\ C_2, (S_2 - DS_2)/S_2, \ldots, (C_n -DC_n)/C_n, (S_n - DS_n)/S_n\}$). We note that each mean and standard deviation are averaged over both DES_SOBER and DES_CBI. From the table, we observe that in 8 programs out of 11, the mean effectiveness of the DES-enabled techniques outperforms that of the respective base techniques.

Table 3
Statistics on relative improvements in effectiveness.

Program	Mean relative improvement in effectiveness (%)	Stdev of relative improvement in effectiveness (%)
print_tokens (2 programs)	145	561
replace	38	170
schedule (2 programs)	94	377
tcas	30	221
tot_info	12	132
flex	−3	31
grep	22	96
gzip	−4	22
sed	−10	94
Weighted average	24	119
Unweighted average	35	189

We also show the weighted averages and unweighted averages for these statistical parameters. The former means averaging the statistical parameters (means or standard deviations) of each program with weights equal to the number of faulty versions of that program. The latter means directly averaging the statistical parameters for each program. In either case, there is, on average, at least a relative increase of 24% in effectiveness by the DES-enabled versions with respect to base techniques SOBER and CBI. However, the effectiveness improvements are not uniform.

Since, on average, there are effectiveness improvements from a base technique to its DES-enabled version, we want to know whether such improvements are statistically significant. We would like to find out the answer to the following hypothesis:

"H_0: *Does a technique enabled with the evaluation sequence approach have no significant difference from the base technique?*"

If the answer is false, we are confident that the DES approach is significantly different from the base fault-localization technique. Considering our previous observation that the DES approach, on average, improves its base version, we may then regard a DES approach as significantly more effective than its base fault-localization technique.

We perform two-tailed Mann–Whitney U-tests to compare the DES-enabled techniques with the corresponding base techniques with respect to every *individual* subject program. The p-values for hypothesis testing on the programs are listed in Table 4.

From the results, we observe that all but one of the p-values are smaller than 0.05 (Dallal, 2007), which indicates that the null hypothesis can be successfully rejected at the 5% significance level. (The only exception is the Mann–Whitney U-test between DES_SOBER and SOBER on grep, which has a *p*-value of 0.09.) In conclusion, the test result of our null hypothesis H_0 implies that DES-enabled techniques are significantly more effective than their base counterparts. Therefore, our answer to RQ2 is that DES-enabled techniques

are significantly more effective than their respective counterparts. The answer to RQ2 also confirms that short-circuit evaluation rules *do* have significant positive impacts on statistical fault localization.[1]

Besides, we also notice that the DES-enabled techniques are marginally less effective than their base counterparts for the gzip and flex programs. One may anticipate that our approach will have more improvements on the base techniques for programs with higher percentages of compound Boolean expressions (as shown in Table 2) than for programs with lower percentages of compound Boolean expressions. This is because our fine-grained approach especially improves the ranking accuracy for compound Boolean expressions. On the other hand, we observe that the DES-enabled techniques perform better on small-sized (Siemens) programs than on medium-sized (UNIX) programs. We notice that the average percentage of compound Boolean expressions (with respect to all expressions) in the UNIX programs is higher than that in the Siemens programs. This unexpected discrepancy can be explained as follows: When we analyze the faults in the Siemens and UNIX programs, we find that higher percentages of faults in the Siemens subjects are on or close to predicate statements, whereas only a few faults in UNIX subjects are on or close to predicate statements. For example, 8 out of 10 faults associated with print_tokens2 are on the Boolean expressions of predicate statements, while only 4 out 17 faults associated with version 3 of flex are on Boolean expressions of predicate statements.

4.3.5. Answering RQ3: Do different evaluation sequences give the same result?

To answer RQ3, we collect the execution statistics of all the evaluation sequences for each Boolean expression in the Siemens suite of programs to calculate the statistical differences between passed and failure-causing test cases. (Owing to our resource limitation, we do not repeat this part of the experiment on the UNIX subject programs.)

We perform a U-test between the evaluation biases for the sets of evaluation sequences over the same predicate in passed and failure-causing test cases. The results of the U-test show that, for 59.12% of the evaluation sequences, there is a significant difference (at the 5% significance level (Dallal, 2007)) between the evaluation biases of passed and failure-causing test cases. In other words, 59.12% of the evaluation sequences are useful fault location indicators, while the remaining 40.87% are not useful standalone fault predicators to differentiate failure-causing test cases from passed ones.

The answer to RQ3 is that different evaluation sequences of the same predicate may have different potentials for fault localization.

4.4. Discussions

Like existing predicate-based fault-localization techniques, our DES technique is also developed on top of predicate evaluation. Unlike them, however, it works at a finer granularity. All such class of techniques (including ours) use predicates to indicate the neighborhoods in which faults may reside. The effectiveness of these techniques generally depends on the locations of the faults and how well predicates surround such faults. In this section, we will elaborate on why DES-enabled techniques are more effective than the respective base techniques by taking a closer look at two important cases in faulty programs. We will further discuss other factors that may affect the performance of our techniques. We will

Table 4
p-Values of *U*-tests on Siemens programs and UNIX programs.

Program	CBI	SOBER
print_tokens (2 programs)	1.04×10^{-15}	2.67×10^{-10}
replace	2.84×10^{-9}	1.89×10^{-16}
schedule (2 programs)	9.80×10^{-11}	3.00×10^{-3}
tcas	4.17×10^{-10}	3.90×10^{-15}
tot_info	1.79×10^{-13}	4.86×10^{-8}
flex	3.18×10^{-4}	4.00×10^{-11}
grep	2.08×10^{-4}	9.08×10^{-2}
gzip	1.57×10^{-27}	4.76×10^{-26}
sed	2.13×10^{-2}	2.23×10^{-2}

---

[1] We are conservative about the conclusion because it is subject to external threats to validity to generalize the results.

also analyze the time complexity and study the empirical performance overheads when applying our technique.

4.4.1. Case 1: Fault on compound predicate

We discuss a fault on a compound predicate in the first case study. The fault is taken from faulty version v9 of the print_tokens2 program. It is in a decision statement on line 218. The code fragments of the original version and the faulty version are shown in Fig. 14.

This fault is caused by adding a Boolean expression headed by an "||" operator to the end of the original compound predicate. The fault will be activated only if the original predicate is evaluated to be false and the extra Boolean expression is evaluated to be true (that is, only if the short-circuit evaluation sequence of the resultant expression is ⟨false, true⟩).

DES-enabled techniques divide test cases into two groups, namely, test cases that exercise the evaluation sequence ⟨false, true⟩ (thus, triggering the fault that leads to a program failures), and test cases that do not trigger the fault. As a result, this evaluation sequence is ranked as highly fault-relevant and its corresponding predicate is deemed to be highly related to the program failure. In our experiment, the rank of the faulty predicate is 10 by DES_SOBER and 11 by DES_CBI.

For the corresponding base techniques, however, test cases with evaluation sequences ⟨true⟩ and ⟨false, true⟩ have been mixed up and treated as similar. As a result, the faulty predicate is perceived by base techniques as less fault-relevant than by DES-enabled techniques. In our experiment, the rank of the faulty predicate is 56 by SOBER and 218 by CBI.

From this case study, we see how a fine-grained analysis technique enables more accurate fault localization.

4.4.2. Case 2: Fault on atomic predicate

Let us further focus on a second case, where a fault is seeded on a predicate having an atomic Boolean expression. Specifically, we take this fault from faulty version v8 of the tot_info program. It is a computational fault seeded to an atomic predicate on line 201, as shown in Fig. 15. For ease of reference, we call this predicate ap_1.

The whole faulty version includes 46 predicates, only two of which contain compound Boolean expressions. We refer to the first one (on line 57) as cp_2 and the second one (on line 308) as cp_3, as listed in Fig. 16. In this example, we use ap_1 to denote an atomic predicate, and cp_2 and cp_3 to denote two compound predicates.

For each of the other 44 atomic Boolean expressions (including ap_1), both CBI and DES_CBI give the same ranking score. The rationale is that the predicates are atomic, and hence there is no possibility of a short-circuit evaluation.

However, CBI gives ranks of 46 and 25 to predicates cp_2 and cp_3, respectively, while our DES_CBI technique gives ranks of 46 and 45, respectively. This is because these two are compound predicates and CBI and DES_CBI may generate different ranking scores (and hence different ranks) for them. Finally, the faulty predicate ap_1 is ranked as 24 by CBI, and ranked as 23 by DES_CBI. Thus, DES_CBI makes a more correct assessment that cp_2 and cp_3 are less fault-relevant than ap_1, whereas CBI mistakenly gives higher suspiciousness to cp_3 than ap_1.

A similar phenomenon is observed for SOBER and DES_SOBER. SOBER gives ranks of 20 and 7 to predicates cp_2 and cp_3, respectively, while DES_SOBER gives ranks of 38 and 41, respectively. For each of the other 44 atomic predicates (including ap_1), both SOBER and DES_SOBER generate the same relative ranking. The faulty predicate ap_1 is ranked as 22 by SOBER and 20 by DES_SOBER. Thus, DES_SOBER makes a more correct assessment that cp_2 and cp_3 are less fault-relevant than ap_1, whereas CBI mistakenly gives higher suspiciousness to cp_2 and cp_3 than ap_1.

In cases where faults are on atomic predicates, there may also exist other predicates that contain compound Boolean expressions. From our previous case study about faults on compound Boolean predicates, we know that DES-enabled techniques may give more accurate ranking results on these compound predicates than SOBER and CBI do. Thus, the noise (possible inaccurate ranking results) from other compound predicates can be reduced. The present case study confirms that DES-enabled techniques may produce a more accurate ranking of predicates even if the faulty predicate is atomic.

4.4.3. Time complexity, actual time-cost, and other discussions

Let $p_1, p_2, \ldots, p_m$ be the Boolean predicates of the program, and $k_1, k_2, \ldots, k_m$ be the numbers of atomic Boolean expressions in the respective predicates. Suppose the time complexity for applying a base technique to investigate one predicate is O_{base}. The time complexity for applying a base technique to the program will be $O(O_{base} \times m)$. The time complexity of the corresponding DES-enabled technique will then be $O(O_{base} \times \sum_{i=1}^{m} k_i)$. This is because a DES-enabled technique uses the same algorithm as the base technique, and the only difference is that the DES-enabled technique works on evaluation sequences while the base technique works on predicates. Thus, for each evaluation of a predicate, in the worst case, it will evaluate all the atomic components of the predicate, and call an invocation of the base algorithm every time.

```
/* Original Version v0 */
if(ch == '\n')

/* Faulty Version v9 */
if(ch == '\n' || ch == '\t')
```

Fig. 14. Code excerpts from versions v0 and v9 of print_tokens2.

```
/* Original Version v0 */
ap₁:  return sum * exp(-x + a * log(x) - LGamma(a))

/* Faulty Version v8 */
ap₁:  return sum * exp(x + a * log(x) - LGamma(a))
```

Fig. 15. Code excerpts from versions v0 and v8 of tot_info.

```
cp₂:  for (p=line; *p != '\0' && isspace((int) *p); ++p)
cp₃:  if (rdf <= 0 || cdf <= 0)
```

Fig. 16. Code excerpts from versions v0 and v8 of tot_info.

Table 5
Timing statistics in the experiment.

Time-cost	DES		Base
	Siemens programs	UNIX programs	Siemens/UNIX Programs
Instrumentation	Comparable to gcc compilation time		
Exe. statistics collection	About 1/10 of program execution time		
Statement ranking	$0.1 \times 10^{-3} - 15.7 \times 10^{-3}$ s	$16.8 \times 10^{-3} - 120.7 \times 10^{-3}$ s	Less than 0.1 s
Locating fault using generated ranking list	$7.0 \times 10^{-6} - 50.0 \times 10^{-6}$ s	$18.0 \times 10^{-6} - 823.0 \times 10^{-6}$ s	Less than 0.1 s

Thus, the time complexity of applying DES on a base technique is higher than that of the base technique. It is easy to figure out that the increase of the time complexity from the base technique to its DES-enable version is $\frac{1}{m}\sum_{i=1}^{m} k_i$. This number is the average number of atomic expressions inside the Boolean expressions in the program. We are confident that it is not a large number in realistic programs. For instance, this number is always smaller than 5 in the Siemens and UNIX programs used in our experiment.

In addition, the data structure of the evaluation sequence needs to be kept during evaluation. What if we translate each Boolean expression into binary code (or lower level representation) and perform a statement-level fault-localization technique on each assembly instruction? Using such a transformation, every atomic component in a compound Boolean expression can be considered, say, as a complete assembly instruction, and the construction of evaluation sequences can be avoided. However, the executions of such instruction statements are not independent of one another, and hence separately estimating their suspiciousness from their execution status may not be accurate. One may further argue to correlate a set of predicates (or statements) to improve the effectiveness of fault identification. We argue, however, that finding such a set of predicates is the exactly basic idea behind our approach. An evaluation sequence contains the information of the legitimate value combinations of atomic predicates that developers compose in the code. We believe that it is a natural and objective criterion to find out such a set of correlating predicates in programs.

What if a technique uses the full combination of truth values of each atomic Boolean expression, but does not consider the evaluation sequences? Suppose $b_1 \oplus b_2 \oplus \cdots \oplus b_n$ (where $\oplus$ stands for a logical operator) is a compound Boolean expression. Since each atomic Boolean expression b_i may have a truth value of either `true` or `false`, the full combination of truth values of these n atomic Boolean expressions is a set of 2^n elements. The time complexity of such a proposal will be $O(O_{base} \times 2^n)$, and some value combinations are very likely to be illegitimate in actual program executions. Consider, for instance, a Boolean expression "`p! = null && p[0]! = null`". The value combination of $\langle$`false`, `true`$\rangle$ cannot appear in any actual program execution owing to the short-circuit evaluation logic in the C language. Compared with the fault indicators above, evaluation sequences of predicates are natural, objective, and effective program entities to extract dynamic features for fault localization.

An empirical study of the actual performances of the DES-enabled techniques compared with those of the respective base techniques is listed in Table 5. The actual time-cost in each step is small enough for practical applicability.

4.5. Threats to validity

We briefly summarize below the threats to validity in our controlled experiment.

Construct validity is related to the platform dependence issues when using the Siemens suite of programs in SIR (Do et al., 2005). Since every program in SIR has a fault matrix file to specify the test verdict of each test case (that is, whether it is a passed or failure-causing test case), we also create a fault matrix file for our test results and carefully verify each test verdict against the corresponding one supplied by SIR. We observe that there are only minor differences in test verdicts between the two fault matrix files. We have thoroughly verified our setting, and believe that the difference is due to platform dependence issues.

Internal validity is related to the risk of having confounding factors that affects the observed results. Following Liu et al. (2006), in the experiment, each technique uses all the applicable test cases to locate fault-relevant predicates in each program. The use of a test suite with a different size may give a different result (Liu et al., 2006). Evaluations on the impact of different test suite sizes on our technique would be welcome. Another important factor is the correctness of our tools. Instead of adopting existing tools used in the literature, we have implemented our own tools in C++ for the purpose of efficiency. To avoid errors, we have adhered to the algorithms in the literature and implemented and tested our tools carefully. To align with previous work, we use the T-score metric to compute the results of this experiment. The use of other metrics may produce different results.

Internal validity is also related to any affecting factors we may or may not have realized. As shown in Section 4.3.4, we have listed the related statistics and explained the reason why our technique appears to be more effective on the small-sized subject programs than the medium-sized subject programs. There may be other implicit factors that may affect the effectiveness of our technique and other predicate-based techniques.

External validity is the degree to which the results can be generalized to test real-world systems. We use the Siemens suite and four UNIX utility programs in the experiment to verify the research questions because they are commonly used by researchers in testing and debugging studies with a view to comparing different work more easily. Further applications of our approach to medium-to-large-sized real-life programs would strengthen the external validity of our work. Each of the faulty versions in our subject programs contains one fault. Despite the competent programmer hypothesis, real-life programs may contain more than one fault. Although Liu et al. (2005) have demonstrated that predicate-based techniques can be used to locate faults in programs that contain more than one fault (Zheng et al., 2006), their effectiveness in this scenario is not well discussed. We will address this threat in future work.

5. Related work

There are rich categories of techniques in statistical fault localization. There are others besides the predicate-based category (Liblit et al., 2005; Liu et al., 2006). Since these two techniques have been explained in previous sections of this paper, we do not repeat the introduction here.

Delta Debugging (Cleve and Zeller, 2005; Zeller and Hildebrandt, 2002) isolates failure-inducing input components, produces cause–effect chains, and locates suspicious faults through the analysis of program state changes during a failed execution

against a successful one. It isolates the relevant variables and values by systematically reducing the state differences between a failed run and a successful one. Other techniques also use a pair of passed test case and failed test case to debug programs. For instance, Renieris and Reiss (2003) find the difference in execution traces between a failed execution and its "nearest neighbor" successful execution. However, coincidental correctness may occur in a successful execution (Wang et al., 2009). A poorly chosen successful run may adversely affect the effectiveness of the above technique and the like. On the other hand, DES may suffer less by having multiple runs to reduce the adverse effect of individual runs and improve the overall reliability.

Other techniques also use the difference in state values for fault localization. For example, Jeffrey et al. (2008) improve Tarantula by considering dataflow information. Their technique collects all possible values of each program variable in all runs, iteratively replaces the value of each program variable in turn in each run, estimates the probability that such replacement converts a failure-causing test case into a passed test case, and then estimates the suspiciousness of statements accordingly. The value replacement concept is interesting. To cast the technique to DES, should a similar replacement idea on atomic predicate evaluation or Boolean expression evaluations be useful? The question has partially been answered by Zhang et al. (2006), in which values of program predicates are switched between `false` and `true` to see whether such a switching converts a failed run into a successful one. Any influencing predicate possibly pinpoints the location of a fault that causes the program to crash. In their work, the test oracle is limited to detecting executions that crash a given program. For failed runs in general, it would be interesting to know whether a fine-grained predicate switching, in the sense of evaluation sequence, can be useful.

DES is a kind of statistical fault-localization technique. Thus, we further compare DES with other representative techniques in this category. Jones et al. (2002) propose Tarantula, which estimates the ratio between the percentages of failure-causing and passed test cases that execute each statement. It then ranks program statements according to their relevance to program faults. A subsequent evaluation (Liu et al., 2005) shows that SOBER and Tarantula cannot outperform each other. In this paper, we have used the same evaluation subjects (the Siemens suite of programs) as in the controlled experiment presented in Liu et al. (2005). Our results show that DES outperforms SOBER on these subject programs. Another difference is that Tarantula ranks individual program entities whereas DES combines multiple fine-gained entities into one during statistical evaluation.

An inherent advantage of producing a ranked sequence of program entities is that developers can conservatively follow the rank to check program entities in turn. However, it is also a limitation due to suppressed parallelism. Jones et al. (2007) use a hierarchical clustering technique to improve the situation. It groups a given set of failed test cases into sub-clusters by using unsupervised learning, and then pairs each cluster of failed test cases with the set of passed test cases when applying Tarantula. DES has not explored parallelism.

Recently, there is further work on empirical evaluation of statistical fault-localization techniques. Yu et al. (2008) examine the impact of test suite reduction on different statement-based fault-localization techniques. Like the controlled experiment results presented in Liu et al. (2005), their results show that reducing the size of test suites have significant deterioration effect on fault localization. On the other hand, Abreu et al. (2009) report that the performance of Tarantula can be largely independent of how test sets are composed and the use of a small number of failed test cases can be very effective in achieving almost optimal diagnosis accuracy. Our DES approach distinguishes different executions that previous

techniques have mixed up. Compared with the base techniques, DES inherently reduces the number of executions applicable to each evaluation sequence. We believe that the result of Abreu et al. (2009) hints a potential way to improve the effectiveness of DES by, for instance, first checking whether a few failed test cases have executed every evaluation sequence, followed by the generation of additional failed test cases for those evaluation sequences that do not involve sufficient failed executions. However, given the mixed results in previous empirical studies, and given that it is more robust to use non-parameter techniques (Zhang et al., 2009), additional empirical studies are needed to confirm our belief.

Studies of test suite sizes highlight the importance of efficiency. Baudry et al. (2006) introduce dynamic basic blocks of a program, which are covered by exactly the same test cases. The lack of variation in the count statistics may not help Tarantula differentiate program statements. They propose a bacteriological approach to finding a subset of a test suite that maximizes the number of dynamic basic blocks. They report that the technique can use fewer test cases than Tarantula to achieve the same effectiveness in fault localization. The same approach may also be applied to DES-enhanced techniques by considering predicates instead of program statements. Intuitively, the concept of maximizing the number of dynamic basic blocks can be adapted to maximize the variations in the predicates or evaluation sequences. We are, however, uncertain before actual experimentation whether such an approach may significantly improve DES.

Another way to reduce cost is to control which test cases to be executed by the program. Jiang et al. (2009) investigate the changes in the effectiveness in fault localization when only a prioritized fragment of a test suite is available. It shows that random test case prioritization to support statistical fault localization can be cost-effective. Their approach is based on selected statement-based fault-localization techniques, whereas DES enhances predicate-based fault-localization techniques. Obviously, more follow-up studies are needed to see whether DES can integrate well with other types of test case prioritization/reduction techniques.

6. Conclusion

Program debugging is time-consuming but important in software development. A major task in debugging is to locate faults. A common approach in statistical fault localization aims at locating program predicates that are close to faulty statements. This relaxes the requirement to pinpoint a fault location and has been shown empirically to be quite effective.

Following this popular trend, we explore a better way to measure the fault relevance of predicates and rank the predicates with respect to their fault relevance. We observe that the fault-localization capabilities of various evaluation sequences of the same Boolean expression are not identical. Because of short-circuit evaluations of Boolean expressions in program execution, different evaluation sequences of a predicate may produce different resultant values. This inspires us to investigate the effectiveness of using Boolean expressions at the evaluation sequence level for statistical fault localization. The experiment on the Siemens programs and UNIX utility programs shows that our approach is promising. Our future work will include locating faults in multi-fault programs using representative test suites, and investigating the effect of coincidental correctness on the performance of our technique. Another future direction is to enhance and develop our work in a continuous integration environment and investigate the result of test suite properties, such as test suite size and coverage information, on the effectiveness of fault-localization techniques.

Acknowledgements

References

Abreu, R., Zoeteweij, P., Golsteijn, R., van Gemund, A.J.C., 2009. A practical evaluation of spectrum-based fault localization. Journal of Systems and Software 82 (11), 1780–1792.

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

Baudry, B., Fleurey, F., Le Traon, Y., 2006. Improving test suites for efficient fault localization. In: Proceedings of the 28th International Conference on Software Engineering (ICSE 2006). ACM Press, New York, NY, pp. 82–91.

Chilimbi, T., Liblit, B., Mehra, K., Nori, A., Vaswani, K., 2009. Holmes: effective statistical debugging via efficient path profiling. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009). IEEE Computer Society Press, Los Alamitos, CA, pp. 34–44.

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

Dallal, G.E., 2007. Why P = 0.05? Available at <http://www.tufts.edu/gdallal/p05.htm>.

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

Hutchins, M., Foster, H., Goradia, T., Ostrand, T., 1994. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: Proceedings of the 16th International Conference on Software Engineering (ICSE '94). IEEE Computer Society Press, Los Alamitos, CA, pp. 191–200.

Jeffrey, D., Gupta, N., Gupta, R., 2008. Fault localization using value replacement. In: Proceedings of the 2008 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008). ACM Press, New York, NY, pp. 167–178.

Jiang, B., Zhang, Z., Tse, T.H., Chen, T.Y., 2009. How well do test case prioritization techniques support statistical fault localization. In: Proceedings of the 33rd Annual International Computer Software and Applications Conference (COMPSAC 2009), vol. 1. IEEE Computer Society Press, Los Alamitos, CA, pp. 99–106.

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

Jones, J.A., Harrold, M.J., Stasko, J., 2002. Visualization of test information to assist fault localization. In: Proceedings of the 24th International Conference on Software Engineering (ICSE 2002). ACM Press, New York, NY, pp. 467–477.

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

Liu, C., Fei, L., Yan, X., Midkiff, S.P., Han, J., 2006. Statistical debugging: a hypothesis testing-based approach. IEEE Transactions on Software Engineering 32 (10), 831–848.

Liu, C., Yan, X., Fei, L., Han, J., Midkiff, S.P., 2005. 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, vol. 30 (5), pp. 286–295.

Renieris, M., Reiss, S.P., 2003. Fault localization with nearest neighbor queries. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003). IEEE Computer Society Press, Los Alamitos, CA, pp. 30–39.

Vessey, I., 1985. Expertise in debugging computer programs: a process analysis. International Journal of Man-Machine Studies 23 (5), 459–494.

Wang, X., Cheung, S.C., Chan, W.K., Zhang, Z., 2009. Taming coincidental correctness: refine code coverage with context pattern to improve fault localization. In: Proceedings of the 31st International Conference on Software Engineering (ICSE 2009). IEEE Computer Society Press, Los Alamitos, CA, pp. 45–55.

Yu, Y., Jones, J.A., Harrold, M.J., 2008. An empirical study of the effects of test-suite reduction on fault localization. In: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008). ACM Press, New York, NY, pp. 201–210.

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

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

Zhang, Z., Chan, W.K., Tse, T.H., Hu, P., Wang, X., 2009. Is non-parametric hypothesis testing model robust for statistical fault localization? Information and Software Technology. doi: 10.1016/j.infsof.2009.06.013.

Zhang, Z., Jiang, B., Chan, W.K., Tse, T.H., 2008. Debugging through evaluation sequences: a controlled experimental study. In: Proceedings of the 32nd Annual International Computer Software and Applications Conference (COMPSAC 2008). IEEE Computer Society Press, Los Alamitos, CA, pp. 128–135.

Zheng, A.X., Jordan, M.I., Liblit, B., Naik, M., Aiken, A., 2006. Statistical debugging: simultaneous identification of multiple bugs. In: Proceedings of the 23rd International Conference on Machine Learning (ICML 2006). ACM Press, New York, NY, pp. 1105–1112.

Zhenyu Zhang is a PhD candidate in computer science at The University of Hong Kong. He received his B.E. and MSc degrees in computer science at Tsinghua University, China. His current research interests are software debugging, software testing and analysis, continuous integration of software engineering activities, and engineering web services and service compositions. He has published his research results in many top-notch international journals and conferences. He received two best paper awards for his COMPSAC 2008 and COMPSAC 2009 papers, and a best paper nominee for his WWW 2009 paper. He is in the program committee of ICSOFT 2009 and SEKE 2010, is a refereed reviewer of *Journal of Systems and Software*, and serves as an external reviewer of many international journals and conferences.

Bo Jiang is a PhD candidate in computer science at The University of Hong Kong. His research interests include debugging and testing of conventional and concurrent software. He has published various conference papers, including two articles that were selected for best paper awards by COMPSAC 2008 and COMPSAC 2009. He also serves as a reviewer of many journals and conferences.

W.K. Chan received his B.E. degree in computer engineering and his MPhil and PhD degrees from The University of Hong Kong in 1993, 1995, and 2004, respectively. He has more than 10 years of industrial experience in developing enterprise systems. From 2005 to 2006, he was a postdoctoral research fellow at Hong Kong University of Science and Technology. He joined City University of Hong Kong in 2006 and is currently an assistant professor. He is on the editorial board of *Journal of Systems and Software*, and is a program chair of QSIC 2010. His current major research interests are to make software testing and analysis techniques for service-based systems, pervasive applications, and general concurrent software sound, effective, efficient, and industrially practical.

T.H. Tse is a professor in computer science at The University of Hong Kong. He received his PhD in Information Systems from the London School of Economics and was a visiting fellow at the University of Oxford. His current research interest is in program testing, debugging, and analysis. He is the steering committee chair of QSIC and an editorial board member of *Journal of Systems and Software; Software Testing, Verification and Reliability; Software: Practice and Experience*; and *Journal of Universal Computer Science*. He is a fellow of the British Computer Society, a fellow of the Institute for the Management of Information Systems, a fellow of the Institute of Mathematics and its Applications, and a fellow of the Hong Kong Institution of Engineers. He was decorated with an MBE by The Queen.

Xinming Wang received his B.E. degree from Zhongshan University, China in 2002 and his M.E. degree from the Chinese Academy of Sciences in 2005. He is currently a PhD candidate at the Hong Kong University of Science and Technology. His research interests include software engineering issues on program analysis, testing, and debugging.