Statistical Debugging: A Hypothesis Testing-Based Approach

Chao Liu, *Member*, *IEEE*, Long Fei, *Member*, *IEEE*, Xifeng Yan, *Member*, *IEEE*, Jiawei Han, *Senior Member*, *IEEE*, and Samuel P. Midkiff, *Member*, *IEEE*

Abstract—Manual debugging is tedious, as well as costly. The high cost has motivated the development of fault localization techniques, which help developers search for fault locations. In this paper, we propose a new statistical method, called SOBER, which automatically localizes software faults without any prior knowledge of the program semantics. Unlike existing statistical approaches that select predicates correlated with program failures, SOBER models the predicate evaluation in both correct and incorrect executions and regards a predicate as fault-relevant if its evaluation pattern in incorrect executions significantly diverges from that in correct ones. Featuring a rationale similar to that of hypothesis testing, SOBER quantifies the fault relevance of each predicate in a principled way. We systematically evaluate SOBER under the same setting as previous studies. The result clearly demonstrates the effectiveness: SOBER could help developers locate 68 out of the 130 faults in the Siemens suite by examining no more than 10 percent of the code, whereas the Cause Transition approach proposed by Holger et al. [6] and the statistical approach by Liblit et al. [12] locate 34 and 52 faults, respectively. Moreover, the effectiveness of SOBER is also evaluated in an "imperfect world," where the test suite is either inadequate or only partially labeled. The experiments indicate that SOBER could achieve competitive quality under these harsh circumstances. Two case studies with grep 2.2 and bc 1.06 are reported, which shed light on the applicability of SOBER on reasonably large programs.

Index Terms—Debugging aids, statistical methods, statistical debugging.

1 Introduction

The last decade has witnessed great advances in fault localization techniques [1], [2], [3], [4], [5], [6], [7], [8], [9]. These techniques aim to assist developers in finding fault locations, which is one of the most expensive debugging activities [10]. Fault localization techniques can be roughly classified as static or dynamic. A static analysis detects program defects by checking the source codes with or without referring to a well-specified program model [1], [2], [3]. A dynamic analysis, on the other hand, typically tries to locate defects by contrasting the runtime behavior of correct and incorrect executions. Dynamic techniques usually do not assume any prior knowledge of program semantics other than the labeling of each execution as either correct or incorrect. Previous studies deploy a variant of program runtime behaviors for fault localization, such as program spectra [11], [4], memory graphs [5], [6], and program predicate evaluation history [7], [12].

Within dynamic analyses, techniques based on predicate evaluations have been shown to be promising for fault localization [13], [14], [7], [12]. Programs are first instrumented with predicates such that the runtime behavior in each execution is encoded through predicate evaluations. Consider the predicate "`idx` $<$ `LENGTH`," where the variable `idx` is an index into a buffer of length `LENGTH`. This predicate checks whether accesses to the buffer ever exceed the upper bound. Statistics on the evaluations of predicates are collected over multiple executions at runtime and analyzed afterward.

The method described in this paper shares the principle of predicate-based dynamic analysis. However, by exploring detailed statistics about predicate evaluation, our method can detect more and subtler faults than the state-of-the-art statistical debugging approach proposed by Liblit et al. [12]. For easy reference, we denote this method as LIBLIT05. For each predicate P in a program $\mathcal{P}$, LIBLIT05 estimates two conditional probabilities:

$$Pr_1 = Pr(\mathcal{P} \text{ fails} | P \text{ is ever observed})$$

and

$$Pr_2 = Pr(\mathcal{P} \text{ fails} | P \text{ is ever observed as } \texttt{true}).$$

It then treats the probability difference $Pr_2 - Pr_1$ as an indicator of how relevant P is to the fault. Therefore, LIBLIT05 essentially regards a predicate fault-relevant if its `true` evaluation correlates with program failures.

While LIBLIT05 succeeded in isolating faults in some widely used software [12], it has a potential problem in its ranking model. Because LIBLIT05 only considers whether a predicate has *ever* been evaluated as `true` or not in each execution, it loses its power to discriminate when a predicate P is observed as `true` at least once in all executions. In this case, Pr_1 is equal to Pr_2, which suggests that the predicate P has no relevance to the fault. In Section 2, we will present an example where the most fault-relevant predicate reveals only a small difference between

- *C. Liu, X. Yan, and J. Han are with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: {chaoliu, xyan, hanj}@cs.uiuc.edu.*
- *L. Fei and S.P. Midkiff are with the School of Electronic and Computer Engineering, Purdue University, West Lafayette, IN 47907. E-mail: {lfei, smidkiff}@purdue.edu.*

Manuscript received 12 Dec. 2005; revised 10 May 2006; accepted 4 Aug. 2006; published online 19 Oct. 2006.
Recommended for acceptance by R. Lutz.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0327-1205.

Pr_1 and Pr_2. We found that similar cases are not rare in practice, as suggested by the experiments in Section 4.

The above issue motivates us to develop a new approach that can exploit multiple evaluations of a predicate within each execution. We start by treating the evaluations of a predicate P as independent Bernoulli trials: Each evaluation of P gives either `true` or `false`. We then estimate the probability of P being `true` in each execution, which we call the *evaluation bias*. While the evaluation bias of P may fluctuate from one execution to another, its observed values from multiple executions constitute a random sample from a statistical model. Specifically, if we let X be the random variable standing for the evaluation bias of predicate P, then there are two statistical models, $f_P(X|Correct)$ and $f_P(X|Incorrect)$, which govern the evaluation bias observed from correct and incorrect executions respectively. Intuitively, if the model $f_P(X|Incorrect)$ is significantly different from $f_P(X|Correct)$, it is indicated that P's evaluation in incorrect runs captures abnormal activity, and the predicate P is likely relevant to the fault. Therefore, instead of selecting predicates correlated with program failures as done by LIBLIT05, our approach statistically models predicate evaluations in both correct and incorrect runs, respectively, and treats the model difference as a measure of the fault relevance.

In quantifying the model difference between $f_P(X|Correct)$ and $f_P(X|Incorrect)$, there are two major obstacles. First, we have no idea what family of distributions the two models are in. Second, we are not authorized to impose model assumptions on $f_P(X)$ because improper model assumptions can result in misleading inferences [15]. Therefore, without prior knowledge of the statistical models, a *direct* measurement of the model divergence is difficult, if not fully impossible.

In this paper, we propose a hypothesis testing-based approach, which *indirectly* quantifies the model difference. Aiming at the model difference, we first propose the null hypothesis that the two models are identical. We then derive a test statistic that conforms to a normal distribution under the null hypothesis through the Central Limit Theorem [15]. Finally, given observed evaluation biases from multiple executions (both correct and incorrect), the instantiated test statistic quantifies the likelihood that the evaluation biases observed from incorrect runs were generated *as if* from $f_P(X|Correct)$. Therefore, a smaller likelihood suggests a larger discrepancy between the two models, and, hence, a greater likelihood that the predicate P is fault-relevant. Using this quantification, we can rank all the instrumented predicates, getting a ranked list of suspicious predicates. Developers can then examine the list from the top down in debugging.

In summary, we make the following contributions in this paper:

1. We propose a probabilistic treatment of program predicates that models how a predicate is evaluated within each execution, which exploits more detailed information than previous methods [7], [12]. In addition, this probabilistic treatment naturally encompasses the concept of program invariants [16] as a special case.

```
01  void subline (char *lin, char *pat, char *sub)
02  {
03    ...
04    while(lin[i] != '\0')
05    {
06      m = amatch(lin, i, pat, 0);
07      if((m >= 0) /* && (lastm != m) */ )
08      {
09          putsub(lin, i, m, sub);
10          lastm = m;
11      }
12      ...
13    }
14  }
```

Fig. 1. Faulty-code version 3 of replace.

2. On top of the probabilistic treatment of predicates, we develop a theoretically well-motivated ranking algorithm, SOBER, that ranks predicates according to how abnormally each predicate evaluates in incorrect executions. Intuitively, the more abnormal the evaluations, the more likely the predicate is fault-relevant.
3. We systematically evaluate the effectiveness of SOBER on the Siemens suite [17], [18] under the same setting as previous studies. Seven existing fault localization techniques are compared with SOBER in this study, which demonstrates the superior accuracy achieved by SOBER in fault localization. Furthermore, the effectiveness of SOBER is also evaluated in an "imperfect world," where the test suite is either inadequate or partially labeled. The experimental results shows that SOBER is statistically robust to these circumstances.
4. Finally, two case studies with grep 2.2 and bc 1.06 are reported, which illustrate the applicability of SOBER on reasonably large programs. In particular, a previously unreported fault is found in bc 1.06, based on the fault localization result from SOBER.

The rest of the paper is organized as follows: Section 2 first presents a motivating example, which illustrates the advantages of modeling predicate evaluations within each execution. We elaborate on the statistical model, ranking the algorithm and its relationship with program invariants in Sections 3. An extensive comparison between SOBER and existing techniques is presented in Section 4, followed by the evaluation of SOBER in an "imperfect world" in Section 5. The two case studies with grep 2.2 and bc 1.06 are reported in Section 6. With related work and threats to validity discussed in Section 7, Section 8 concludes this study.

2 A Motivating Example

In this section, we present a detailed example that illustrates the advantage of modeling predicates in a probabilistic way. This example inspires us to locate faults by quantifying the divergence between the models of correct and incorrect executions.

The program in Fig. 1 is excerpted from the third faulty version of the replace program in the Siemens suite. The program replace has 507 lines of C code (LOC) and it performs regular expression matching and substitutions. The second subclause in line 7 was intentionally commented out by the Siemens researchers to simulate a type of

	A	$\neg A$
B	*enter*	*skip*
$\neg B$	*enter*	*skip*

(a)

	A	$\neg A$
B	*enter*	*skip*
$\neg B$	*skip*	*skip*

(b)

Fig. 2. Branching actions in (a) $\mathcal{P}$ and (b) $\widehat{\mathcal{P}}$.

	A	$\neg A$
B	n_{AB}	$n_{A\bar{B}}$
$\neg B$	$n_{A\bar{B}} = 0$	$n_{\bar{A}\bar{B}}$

(a)

	A	$\neg A$
B	n'_{AB}	$n'_{A\bar{B}}$
$\neg B$	$n'_{A\bar{B}} \geq 1$	$n'_{\bar{A}\bar{B}}$

(b)

Fig. 3. (a) A correct and (b) an incorrect execution in $\mathcal{P}$.

fault that may sneak in if the developer fails to think fully about the `if` condition. Since this is essentially a logic error that does not incur program crashes, even experienced developers would have to use a conventional debugger for step-by-step tracing. Our question is: *Can we guide developers to the faulty location or its vicinity by contrasting the runtime behaviors between correct and incorrect executions?*

For clarity in what follows, we denote the program with the subclause (`lastm != m`) commented out as the incorrect (or faulty) program $\mathcal{P}$, and the one with the subclause (i.e., (`lastm != m`) is not commented out) as the correct program $\widehat{\mathcal{P}}$. Because $\widehat{\mathcal{P}}$ is certainly not available when $\mathcal{P}$ is debugged, $\widehat{\mathcal{P}}$ is used here to illustrate how our method is motivated. As shown in Section 3, our method collects statistics only from the faulty program $\mathcal{P}$, *not* from $\widehat{\mathcal{P}}$.

We declare two Boolean variables, A and B, as follows:

$$\texttt{A} = (\texttt{m} >= 0);$$
$$\texttt{B} = (\texttt{lastm} != \texttt{m});$$

Let us consider the four possible evaluation combinations of A and B, and their corresponding branching actions (either *enter* or *skip* the block from lines 8 through 11) in both $\mathcal{P}$ and $\widehat{\mathcal{P}}$. Fig. 2 explicitly lists the actions in $\mathcal{P}$ (Fig. 2a) and $\widehat{\mathcal{P}}$ (Fig. 2b). The left panel shows the actual actions taken in the faulty program $\mathcal{P}$, while the right one lists the expected actions in $\widehat{\mathcal{P}}$.

Differences between the above two tables reveal that in the faulty program $\mathcal{P}$, unexpected actions take place if and only if $A \wedge \neg B$ evaluates to `true`. Explicitly, when $A \wedge \neg B$ is `true`, the control flow actually enters the block, whereas it is expected to skip the block if the logic was correct. This incorrect control flow will likely lead to incorrect outputs. Therefore, for the faulty program $\mathcal{P}$, an execution is incorrect if and only if there exist `true` evaluations of $A \wedge \neg B$ at line 7; otherwise, the execution is correct even though the program contains a fault.

While the predicate $P : (A \wedge \neg B) = \texttt{true}$ precisely characterizes the scenario under which incorrect executions take place, there is little chance for any fault locator to spot P as fault-relevant. The obvious reason is that while we are debugging $\mathcal{P}$, $\widehat{\mathcal{P}}$ is not available. Therefore, we have no idea of what B is, let alone its combination with A. On the other hand, because the evaluation of A is observable in $\mathcal{P}$, we are interested in whether the evaluation of A can actually point to the fault. Explicitly, if the evaluation of A in incorrect executions significantly diverges from that in correct ones, the `if` statement at line 7 may be regarded as fault-relevant, which exactly points to the fault location.

We, therefore, contrast how A is evaluated differently in correct and incorrect executions of $\mathcal{P}$. Fig. 3 shows the number of `true` evaluations for the four combinations of A and B in one correct (Fig. 3a) and one incorrect (Fig. 3b) execution. The major difference between the two is that in a correct run, $A \wedge \neg B$ never evaluates `true` ($n_{A\bar{B}} = 0$) while $n'_{A\bar{B}}$ must be nonzero for an execution to be incorrect. Since the `true` evaluation of $A \wedge \neg B$ implies $A = \texttt{true}$, we expect that the probability for A to be `true` is different in correct and incorrect executions. In running 5,542 test cases, the `true` evaluation probability is 0.2952 in a correct execution and 0.9024 in an incorrect execution, on average. This divergence suggests that the fault location (i.e., line 7) does exhibit detectable abnormal behaviors in incorrect executions. Our method, as described in Section 3, nicely captures this divergence and ranks $A = \texttt{true}$ as the top fault-relevant predicate. This predicate readily leads the developer to the fault location. Meanwhile, we note that because neither $A = \texttt{true}$ nor $A = \texttt{false}$ is an invariant in correct or incorrect executions, invariant-based methods cannot detect that A is a suspicious predicate. LIBLIT05 does not regard A as suspicious either because it does not model the predicate evaluation *within* each execution (see Section 3.7 for details).

The above example illustrates a simple but representative case where a probabilistic treatment of predicates captures detailed information about predicate evaluations. In the next section, we describe the statistical model and the ranking algorithm that implement this intuition.

3 Predicate Ranking Models

3.1 Problem Settings

Let $T = \{t_1, t_2, \cdots, t_n\}$ be a test suite for program $\mathcal{P}$. Each test case $t_i = (d_i, o_i)$ $(1 \leq i \leq n)$ has an input d_i and the expected output o_i. The execution of $\mathcal{P}$ on each test case t_i gives the output $o'_i = \mathcal{P}(d_i)$. We say $\mathcal{P}$ passes the test case t_i (i.e., t_i is a passing case) if and only if o'_i is identical to o_i; otherwise, $\mathcal{P}$ fails on t_i (i.e., t_i is a failing case). In this way, the test suite T is partitioned into two disjoint subsets T_p and T_f, corresponding to the passing and failing cases respectively:

$$T_p = \{t_i | o'_i = \mathcal{P}(d_i) \text{ and } o'_i = o_i\},$$
$$T_f = \{t_i | o'_i = \mathcal{P}(d_i) \text{ and } o'_i \neq o_i\}.$$

Since program $\mathcal{P}$ passes test case t_i if and only if $\mathcal{P}$ executes correctly, we use "correct" and "passing," as well as "incorrect" and "failing," interchangeably in the following discussion.

Given a faulty program $\mathcal{P}$ together with a test suite $T = T_p \cup T_f$, our task is to *localize the suspicious fault region by contrasting $\mathcal{P}$'s runtime behaviors on T_p and T_f.*

3.2 Probabilistic Treatment of Predicates

In general, a program predicate is a proposition about any program property, such as "`idx` $<$ `LENGTH`," "`!empty(list)`," and "`foo()` $>$ `0`." As any instrumentation site can be touched more than once due to program control flows, a predicate P can be evaluated multiple times in one execution, and each evaluation produces either `true` or `false`. In order to model this *within-execution* behavior of P, we propose the concept of evaluation bias, which estimates the probability of the predicate P being evaluated as `true`.

Definition 1 (Evaluation Bias). *Let n_t be the number of times that predicate P evaluates to* `true`*, and n_f the number of times it evaluates to* `false`*, in one execution. $\pi(P) = \frac{n_t}{n_t+n_f}$ is the observed evaluation bias of predicate P in this particular execution.*

Intuitively, $\pi(P)$ estimates the probability that P takes the value `true` in each evaluation. If the instrumentation site of P is touched at least once (i.e., $n_t + n_f \neq 0$), $\pi(P)$ varies in the range of [0, 1]: $\pi(P)$ is equal to 1 if P always holds, to 0 if it never holds, and in between for all other sets of outcomes. If the predicate is never evaluated, $\pi(P)$ has a singularity 0/0. In this case, since we have no evidence to favor either `true` or `false`, we set $\pi(P)$ to 0.5 for fairness. Finally, if a predicate is never evaluated in any failing runs, it has nothing to do with program failures and is hence eliminated from the predicate ranking.

3.3 Methodology Overview

We formulate the main idea of our method in this section and then develop its details in Section 3.4. Following the convention in statistics, we use uppercase letters for random variables and the corresponding lowercase letters for their realizations. Moreover, $f(X|\theta)$ is a general notation of the probability model for the random variable X that is indexed by the parameter θ.

Let the entire test case space be $\mathcal{T}$, which conceptually contains all the possible inputs and expected outputs. According to the correctness of $\mathcal{P}$ on the test cases in $\mathcal{T}$, $\mathcal{T}$ can be partitioned into two disjoint sets $\mathcal{T}_p$ and $\mathcal{T}_f$ for passing and failing cases. Therefore, the available test suite T and its partitions T_p and T_f can be treated as a random sample from $\mathcal{T}$, $\mathcal{T}_p$, and $\mathcal{T}_f$, respectively. Let X be the random variable for the evaluation bias of predicate P. We then use $f_P(X|\theta_p)$ and $f_P(X|\theta_f)$ to denote the statistical model for the evaluation bias of P in $\mathcal{T}_p$ and $\mathcal{T}_f$, respectively. Therefore, the evaluation bias from running a test case t can be treated as an observation from $f_P(X|\theta)$, where θ is either θ_p or θ_f depending on whether t is passing or failing. Given the statistical models for both passing and failing runs, we then define the fault relevance of P as follows:

Definition 2 (Fault Relevance). *A predicate P is relevant to the hidden fault if its underlying model $f_P(X|\theta_f)$ diverges from $f_P(X|\theta_p)$, where X is the random variable for the evaluation bias of P.*

The above definition relates $f_P(X|\theta)$, the statistical model for P's evaluation bias, to the hidden fault. Naturally, the larger the difference between $f_P(X|\theta_f)$ and $f_P(X|\theta_p)$, the more relevant P is to the fault. Let $\mathbf{L}(P)$ be an arbitrary similarity function,

$$\mathbf{L}(P) = \mathbf{Sim}(f(X|\theta_p),\ f(X|\theta_f)). \tag{1}$$

The ranking score $s(P)$ can be defined as $g(\mathbf{L}(P))$, where $g(x)$ can be any monotonically decreasing function. We here choose $g(x) = -log(x)$ because $log(x)$ effectively measures the relative magnitude even when xs are closed to 0 (certainly, x must be positive). Therefore, the fault relevance score $s(P)$ is defined as

$$s(P) = -log(\mathbf{L}(P)). \tag{2}$$

Using this fault relevance score, we can rank all the instrumented predicates, and the top-ranked ones are regarded more likely to be fault-relevant. Therefore, the fault localization problem boils down to the setting of the similarity function, which, in turn, consists of two subproblems: 1) What is a suitable similarity function $\mathbf{L}(P)$, and 2) how is $\mathbf{L}(P)$ computed when the closed form of $f_P(X|\theta)$ is unknown? In Sections 3.4 and 3.5, we examine the two problems in detail.

3.4 Predicate Ranking

The lack of prior knowledge about $f_P(X|\theta)$ constitutes one of the major obstacles in calculating the similarity (or difference, equivalently) between $f_P(X|\theta_p)$ and $f_P(X|\theta_f)$. If the closed form of $f_P(X|\theta_p)$ and $f_P(X|\theta_f)$ were given, measures used in information theory [19], such as the relative entropy, would immediately apply. Meanwhile, we are not authorized to impose model assumptions, like normality, on $f_P(X)$ because improper assumptions can lead to misleading inferences. Therefore, given the above difficulties in directly measuring the model difference, in this paper we propose an indirect approach that measures the difference between $f_P(X|\theta_p)$ and $f_P(X|\theta_f)$ without any model assumption.

Aiming at the model difference, we first propose the *null hypothesis* that $\mathcal{H}_0 : f_P(X|\theta_p) = f_P(X|\theta_f)$, i.e., there is no difference between the two models. Letting $\mathbf{X} = (X_1, X_2, \cdots, X_m)$ be a random sample from $f_P(X|\theta_f)$ (i.e., observed evaluation bias from m failing cases), we derive a statistic Y, which, under the null hypothesis $\mathcal{H}_0$, conforms to a known distribution. If the realized statistic $Y(\mathbf{X})$ corresponds to an event that has a small likelihood of happening, the null hypothesis $\mathcal{H}_0$ is likely invalid, which suggests that a nontrivial difference exists between $f_P(X|\theta_p)$ and $f_P(X|\theta_f)$.

We choose to characterize $f_P(X|\theta)$ through its population mean μ and variance σ^2, so that the null hypothesis $\mathcal{H}_0$ is

$$\mu_p = \mu_f \text{ and } \sigma_p^2 = \sigma_f^2. \tag{3}$$

Let $\mathbf{X} = (X_1, X_2, \cdots, X_m)$ be an *independent and identically distributed (i.i.d.)* random sample from $f_P(X|\theta_f)$. Under the null hypothesis, we have $E(X_i) = \mu_f = \mu_p$ and $Var(X_i) = \sigma_f^2 = \sigma_p^2$. Because $X_i \in [0, 1]$, both $E(X_i)$ and $Var(X_i)$ are finite. According to the Central Limit Theorem [15], the following statistic

$$Y = \frac{\sum_{i=1}^{m} X_i}{m}, \tag{4}$$

asymptotically conforms to $N(\mu_p, \frac{\sigma_p^2}{m})$, a normal distribution with mean μ_p and variance $\frac{\sigma_p^2}{m}$.

Let $f(Y|\theta_p)$ be the probability density function of the normal distribution $N(\mu_p, \frac{\sigma_p^2}{m})$. Then, the likelihood $L(\theta_{qp}|Y)$ of θ_p given the observed Y is

$$L(\theta_p|Y) = f(Y|\theta_p). \tag{5}$$

A smaller likelihood implies that $\mathcal{H}_0$ is less likely to hold, which, in turn, indicates a larger difference between $f_P(X|\theta_p)$ and $f_P(X|\theta_f)$. Therefore, we can reasonably instantiate the similarity function in (1) with the likelihood function

$$\mathbf{L}(P) = L(\theta_p|Y). \tag{6}$$

According to the property of normal distribution, the normalized statistic

$$Z = \frac{Y - \mu_p}{\sigma_p/\sqrt{m}} \tag{7}$$

asymptotically conforms to the standard normal distribution $N(0,1)$, and

$$f(Y|\theta_p) = \frac{\sqrt{m}}{\sigma_p}\varphi(Z), \tag{8}$$

where $\varphi(Z)$ is the probability density function of $N(0,1)$.

Combining (2), (6), (5), and (8), we finally get the fault-relevance ranking score for predicate P as

$$s(P) = -log(\mathbf{L}(P)) = log\left(\frac{\sigma_p}{\sqrt{m}\varphi(Z)}\right). \tag{9}$$

3.5 Discussions on Score Computation

First, in order to calculate $s(P)$ using (9), we need to estimate the population mean μ_p and the standard error σ_p of $f_P(X|\theta_p)$. Let $\mathbf{X}' = (X_1', X_2', \cdots, X_n')$ be a random sample from $f_P(X|\theta_p)$ (which corresponds to the observed evaluation bias from the n passing runs), then μ_p and σ_p can be estimated as

$$\mu_p = \overline{X'} = \frac{\sum_{i=1}^{n} X_i'}{n} \tag{10}$$

and

$$\sigma_p = S_{X'} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(X_i' - \overline{X'})^2}. \tag{11}$$

Second, because the $\sqrt{m}$ in (9) does not affect the relative order between predicates, it can be safely dropped in practice. However, as simple algebra would reveal, the m in (4) and the $\sqrt{m}$ in (7) cannot be discarded, because they properly scale the statistics for standard normality as required by the Central Limit Theorem.

Finally, we note that although the derivation of (9) is based on the asymptotic behavior, i.e., when $m \to +\infty$, statistical inference suggests that the asymptotic result is still valid even when the sample size is nowhere near infinity [15]. In the fault localization scenario, it is true that we cannot have an infinite number of failing cases. But as shown in experiments, (9) still works well in ranking abnormal predicates even when only a small number of failing cases are available.

We now use a concrete example to conclude the discussion in this subsection. The example illustrates how the fault relevance score of the predicate $P = (A = \texttt{true})$ is calculated for the program in Fig. 1.

First, by running the 130 failing and the 5,412 passing cases (i.e., $m = 130$ and $n = 5{,}412$) on the instrumented program, the numbers of `true` and `false` evaluations are recorded at runtime for each execution. Then, the evaluation bias of P in each execution is calculated based on Definition 1. Next, the statistic $Y = 0.9024$ is directly obtained from the evaluation biases in failing cases according to (4). Similarly, from passing cases, we get $\mu_p = 0.2952$ and $\sigma_p = 0.2827$ according to (10) and (11), respectively. Plugging the calculated Y, μ_p, σ_p and $m = 130$ into (7), we get $Z = 24.4894$. Finally, from (9), the fault relevance score for predicate P is 297.2.

Besides illustrating how $s(P)$ is calculated, this example also shows the role played by the log operator in (9). Although the log operator does not influence the ranking of predicates, it helps scale down the calculated score, which might otherwise overflow in numeric computation.

3.6 Generalizing Invariants

In this section, we demonstrate how the probabilistic treatment of predicate evaluations encompasses program invariants [16] as a special case. Moreover, we also prove that the fault relevance score in (9) readily identifies both invariant violations and conformations.

Without loss of generality, a predicate P is a program invariant on a test suite C if and only if it always evaluates `true` during the execution of C. In practice, the test suite C is usually chosen to be a set of passing cases so that the summarized invariants characterize the correct behavior of the subject program [16]. During the failing executions, these invariants are either conformed (i.e., still evaluate `true`) or violated (i.e., evaluate `false` at least once), and those violated invariants are regarded as hints for debugging. In some special cases, the test suite C is chosen to be a time interval during which the execution is believed to be correct. One typical example is that for software that runs for a long time, such as Web servers, the execution is likely correct at the beginning of the execution [14].

According to Definition 1, the evaluation bias of an invariant is always 1. Taking the set of passing cases T_p as C, we know that, if the predicate P is an invariant, $\mu_p = 1$ and $\sigma_p = 0$. Moreover, the following theorem proves that the fault relevance score function of (9) naturally identifies both invariant violations and conformations.

Theorem 1. *Let P be any invariant summarized from a set of correct executions T_p. $s(P) = +\infty$ if P is violated in at least one faulty execution, and $s(P) = -\infty$ if P is conformed in all faulty executions.*

Proof. Let $\mathbf{x} = (x_1, x_2, \cdots, x_m)$ be a realized random sample, which corresponds to the observed evaluation biases from the m failing runs. Once P is violated in at least one execution, $\sum_{i=1}^{m} x_i \neq m$. It then follows from (7) that

$$z = \frac{c}{\sigma_p}, \text{ where } c = \frac{\sum_{i=1}^{m} x_i - m\mu_p}{\sqrt{m}} \neq 0,$$

and then

$$\lim_{\sigma_p \to 0} \frac{\sigma_p}{\sqrt{m}\varphi(z)} = \sqrt{\frac{2\pi}{m}} \lim_{\sigma_p \to 0} \frac{\sigma_p}{e^{-\frac{1}{2}\left(\frac{c}{\sigma_p}\right)^2}} = \sqrt{\frac{2\pi}{m}} \lim_{t \to \infty} \frac{e^{\frac{c^2 t^2}{2}}}{t}$$

$$= c^2 \sqrt{\frac{2\pi}{m}} \lim_{t \to \infty} t e^{\frac{c^2 t^2}{2}} = +\infty.$$

Thus, (9) gives $s(P) = +\infty$. This means that SOBER treats violated invariants as the most abnormal predicates and ranks them at the top.

On the other hand, if the invariant P is not violated in any failing execution, we have

$$\lim_{\sigma_p \to 0} z = \lim_{\sigma_p \to 0} \frac{\sum_{i=1}^{m} x_i - m\mu_p}{\sqrt{m}\sigma_p} = \lim_{\sigma_p \to 0} \frac{0}{\sqrt{m}\sigma_p} = 0,$$

and, therefore,

$$\lim_{\sigma_p \to 0} \frac{\sigma_p}{\sqrt{m}\varphi(z)} = \lim_{\sigma_p \to 0} \frac{\sigma_p}{\sqrt{m}\varphi(0)} = 0,$$

which immediately leads to $s(P) = -\infty$. This suggests that conformed invariants are regarded as the least abnormal, and are ranked at the bottom by our method. □

Theorem 1 indicates that, if a fault can be caught by invariant violations as implemented in the DIDUCE [14] project, SOBER can also detect it because the fault relevance score for the violated invariant is $+\infty$. Meanwhile, for conformed invariants, SOBER simply discards them due to the $-\infty$ score. Previous research suggests that invariant violations by themselves can only locate a number of faults in the Siemens suite [20]. As will be shown shortly, our method SOBER, being a superset of invariant-based methods, actually achieves the best fault localization results on the Siemens suite.

3.7 Differences between SOBER and LIBLIT05

Because both LIBLIT05 and SOBER are based on a statistical analysis of predicate evaluations, we now illustrate the differences in this section.

In principle, LIBLIT05 contrasts the probability that one execution crashes if the predicate P is *ever* observed `true`, and that if P is observed (either `true` or `false`) in the execution. Specifically, the authors define

$$Context(P) = Pr(Crash|P\ observed), \quad (12)$$
$$Failure(P) = Pr(Crash|P\ observed\ \texttt{true}), \quad (13)$$

and take the probability difference

$$Increase(P) = Failure(P) - Context(P) \quad (14)$$

as one of the two key components of P's fault relevance score. The other component is the number of failing runs where P is ever observed as `true`. A harmonic average is then taken to combine these two components.

A detailed examination reveals fundamental differences between LIBLIT05 and SOBER. First, from the methodological point of view, LIBLIT05 estimates how much more likely an execution crashes if the predicate P is observed as `true` in comparison with if P is observed as either `true` or `false`. This indicates that LIBLIT05 places a greater value on predicates whose `true` evaluation correlates with program crashes. SOBER, on the other hand, models the evaluation distribution of the predicate P in passing (i.e., $f_P(X|\theta_p)$) and failing (i.e., $f_P(X|\theta_f)$) executions, respectively, and regards predicates with large differences between $f_P(X|\theta_f)$ and $f_P(X|\theta_p)$ as fault-relevant. Therefore, SOBER and LIBLIT05 actually follow two fundamentally different approaches, although both of them rank predicates statistically. Secondly, SOBER explores the multiple evaluations of predicates *within one execution* while LIBLIT05 overlooks this information. For instance, if a predicate P evaluates as `true` at least once in each execution, and has different likelihood to be `true` in passing and failing executions, LIBLIT05 simply overlooks P while SOBER can readily capture the evaluation divergence.

Let us reexamine the program in Fig. 1 presented in Section 2. The faulty statement (line 7) is executed in almost every execution. Within each run, it evaluates multiple times as either `true` or `false`. In this case, LIBLIT05 has little discrimination power. Specifically, for the predicate

$$P : "(m\ >=\ 0) = \texttt{true}",$$

$Increase(P) = 0.0104$, and the $Increase$ value for predicate

$$P' : "(m\ >= 0) = \texttt{false}"$$

is -0.0245. According to [12], neither P nor P' is ranked on top since they are either negative or too small. Thus, LIBLIT05 fails to identify the fault. In comparison, SOBER successfully ranks the predicate P as the most suspicious predicate. Intuitively, this is because the evaluation bias in failing executions (0.9024) significantly diverges from that in passing ones (0.2952).

4 Empirical Comparison with Existing Techniques

In this section, we empirically evaluate the effectiveness of SOBER in fault localization. We compare SOBER with seven existing fault localization algorithms under the same setting as previous studies. Section 4.1 first describes the experimental setup, which includes the subject programs, the metric for localization quality, and the implementation details. We briefly explain the seven fault localization algorithms in Section 4.2. Detailed comparison results are presented in Sections 4.3 and 4.4. Finally, Section 4.5 compares these algorithms from different perspectives other than the localization accuracy.

4.1 Experimental Setup

In this study, we use the Siemens suite as the subject programs. The Siemens suite was originally prepared by Siemens Corp. Research in a study of test adequacy criteria [17]. It contains 132 faulty versions of seven subject programs, where each faulty version contains one and only one manually injected fault. Table 1 lists the characteristics of the seven subject programs. The medians of the failing and passing cases are taken over all the faulty versions of each subject program. Readers interested in more details about the Siemens suite are referred to [17], [18].

Previously, many researchers investigating fault localization have reported their results on the Siemens suite [11], [20], [4], [6]. Because no failures are observed for the 32nd version of the replace program and the 10th version of the schedule2 program on the accompanying test suites, these two versions are excluded in previous studies [4], [6], [21], as well as in this one.

In order to objectively quantify the localization accuracy, an evaluation framework based on program static dependencies is adopted in this study. This measure was

TABLE 1
Characteristics of Subject Programs

	Faulty Versions	LOC	Total Test Cases	Median of Failing Cases	Median of Passing Cases
print_tokens	7	539	4130	38	4082
print_tokens2	10	489	4115	223	3892
replace	32	507	5542	83	5459
schedule	9	397	2650	31	2619
schedule2	10	299	2710	32	2678
tcas	41	174	1608	23	1585
tot_info	23	398	1052	71	981

originally proposed by Renieris and Reiss [4], and was later adopted by Cleve and Zeller in reporting the quality of CT [6]. We briefly summarize this measure as follows:

1. Given a (faulty) program $\mathcal{P}$, its program dependence graph (PDG) is written as $\mathcal{G}$, where each statement is a vertex and there is an edge between two vertices if two statements have data and/or control dependencies.
2. The vertices corresponding to faulty statements are marked as *defect* vertices. The set of defect vertices is written as V_{defect}.
3. Given a fault localization report R, which is a set of suspicious statements, their corresponding vertices are called *blamed* vertices. The set of blamed vertices is written as V_{blamed}.
4. A developer can start from V_{blamed} and perform a breadth-first search until he reaches one of the defect vertices. The set of statements covered by the breadth-first search is written as $V_{examined}$.
5. The T-score, defined as follows, measures the percentage of code that has been examined in order to reach the fault:

$$T = \frac{|V_{examined}|}{|V|} * 100\%, \tag{15}$$

where $|V|$ is the size of the program dependence graph $\mathcal{G}$. In [4], [6], the authors used $1 - T$ as an equivalent measure.

The T-score estimates the percentage of code a developer needs to examine (along the static dependencies) before the fault location is found, when a fault localization report is provided. A high quality fault localization is expected to be a small set of statements that are close to (or contain) the fault location. The above definition of T-score is immediately applicable to localizations that consist of a *set* of "blamed" statements. For algorithms that generate a ranked list of all predicates, like LIBLIT05 and SOBER, the corresponding statements of the top k predicates are taken as a fault localization report. The optimal k is the one that minimizes the average examined code over a set of faults under study, i.e.,

$$k_{opt} = \arg\min_{k} \ \mathrm{E}[T_k]. \tag{16}$$

where $\mathrm{E}[T_k]$ is the average T-score for the given set of faults for any fixed k.

As the above defined T-score is calculated based on PDGs, we call it *PDG-based*. Recently, another kind of T-score was used by Jones and Harrold in reporting the localization results of TARANTULA [8]. The TARANTULA tool produces a ranking of *all* executable statements, and the authors calculate the T-score directly from the ranking. Instead of taking the top k statements and calculating the T-score based on PDGs, the authors examine whether the faulty statements are ranked high. Specifically, a developer is assumed to examine statement by statement from the top of the ranking until a faulty statement is touched. The percentage of statements examined by then is taken as the T-score. We call the T-score calculated in this way *ranking-based*. Apparently, the ranking-based T-score assumes a different code examination strategy than that assumed by the PDG-based, i.e., along the ranking rather than along the dependencies. Intuitively, the PDG-based approach is closer to practice. Moreover, the ranking-based T-score is not as generally applicable as the PDG-based, because it requires a ranking of *all* statements. For example, none of the discussed algorithms in Section 4.2, except TARANTULA, can be evaluated using the ranking-based approach, but TARANTULA *can* be evaluated by the PDG-based T-score by taking the top k statements as a fault localization report.

In this study, we compare SOBER with seven existing fault localization algorithms (described in the next section). Among them, we implemented LIBLIT05 in Matlab and validated the correctness of the implementation with the original authors. For the other six algorithms, the localization result on the Siemens suite is taken directly from their corresponding publications.

We instrumented the subject programs with two kinds of predicates: branches and function returns, which are described in detail in [7], [12]. In particular, we treat each branch conditional as one inseparable instrumentation unit, and do not consider each subclause separately. For better fault localization, one may be tempted to introduce more predicates. But the introduction of more predicates is a double-edged sword. On the positive side, an expanded set of predicates is more likely to cover the faulty code; but the superfluous predicates brought in can nontrivially complicate the predicate ranking. So far, no agreement has been reached on what are the "golden predicates." At runtime, the evaluation of predicates is collected without sampling for both LIBLIT05 and SOBER.

All experiments in this section were carried out on a 3.2 GHz Intel Pentium-4 PC with 1 GB physical memory, running Fedora Core 2. In calculating the T-scores, we used CODESURFER 1.9 with patch 3 to generate the program dependence graphs. Because PDGs generated by CODESURFER may vary with different build options, the factory default (by enabling the `factory-default` switch) is used to allow reproducible results in the future. Moreover, the Matlab source code of SOBER and the instrumented Siemens suite are available online at http://www.ews.uiuc.edu/~chaoliu/sober.htm.

4.2 Compared Fault Localization Algorithms

We now briefly explain the seven fault localization algorithms we compare with SOBER. As LIBLIT05 is already discussed in Section 3.7, we only describe the other six algorithms below:

- **Set-Union**. This algorithm is based on the program spectra difference between a failing case f and a set of passing cases P. Specifically, let $S(t)$ be the program spectra of running the test case t. Then, the set difference between $S(f)$ and the *union* spectra of cases in P is taken as the fault localization report R, i.e., $R = S(f) - \cup_{p_i \in P} S(p_i)$. This algorithm is described in [4], and we denote it by UNION for brevity.
- **Set-Intersect**. A complementary algorithm to UNION is also described in [4]. It is based on the set difference between the spectra of the failing case and the *intersection* spectra of passing cases, namely, the localization report $R = \cap_{p_i \in P} S(p_i) - S(f)$. We denote this algorithm by INTERSECT.
- **Nearest Neighbor**. The nearest neighbor approach, proposed by Renieris and Reiss in [4], contrasts the failing case to the passing case that most "resembles" the failing case. Namely, the localization report $R = S(f) - S(p)$, where p is the nearest passing case to f as measured under certain distance metrics. The authors studied two distance metrics and found that the nearest neighbor search based on the Ulam's distance renders better fault localization. This algorithm is denoted as NN/PERM by the original authors.
- **Cause Transition**. The Cause Transition algorithm [6], denoted as CT, is an enhanced variant of Delta Debugging [5]. Delta Debugging contrasts the memory graph [22] of one failing execution, e_f, against that of one passing execution, e_p. By carefully manipulating the two memory graphs, Delta Debugging systematically narrows the difference between e_f and e_p down to a small set of suspicious variables. CT enhances Delta Debugging by exploiting the notion of *cause transition*: "moments where new relevant variables begin being failure causes" [6]. Therefore, CT essentially implements the concept of "search in time" in addition to the original "search in space" used in Delta Debugging.
- **Tarantula**. The TARANTULA tool was originally presented to visualize the test information for each statement in a subject program, and it was shown to be useful for fault localization [23]. In a recent study [8], the authors took $(1 - hue(s))$ as the fault relevance score for the statement s, where $hue(s)$ is the hue component of each statement in visualization [23]. With the fault relevance score calculated for each statement, TARANTULA produces a ranking of all executable statements. Developers are expected to examine the ranking from the top down to locate the fault.
- **Failure-Inducing Chops**. Gupta et al. recently propose a fault localization algorithm that integrates delta debugging and dynamic slicing [9]. First, a minimal *failure-inducing input* f' is derived from the given failing case f using the algorithms of Zeller and Hildebrandt [24]. Then, a forward dynamic slice, FS, and a backward slice, BS, are calculated from f' and the erroneous output, respectively. Finally, the intersection of FS and BS, i.e., the chop, is taken as the fault localization report, namely, $R = FS \cap BS$. We denote this algorithm by SLICECHOP.

In previous studies, comparisons of some of the above algorithms are reported. Specifically, Renieris and Reiss found that NN/PERM outperformed both UNION and INTERSECT [4], whereas Cleve and Zeller later reported that a better result than NN/PERM was achieved by CT [6]. These reported results are all based on the PDG-based T-score. As CT achieves the best localization result as measured with the PDG-based T-score, we compare SOBER with CT and LIBLIT05 in Section 4.3 using the same measure. Because TARANTULA and SLICECHOP results are not reported with the PDG-based T-score, we compare SOBER with them separately in Section 4.4.

4.3 Comparison with LIBLIT05 and CT

In this section, we compare SOBER with CT and LIBLIT05. We subject both LIBLIT05 and SOBER to the 130 faults in the Siemens suite and measure their localization quality using the PDG-based T-score (15). The result of CT is directly cited from [6].

Fig. 4a depicts the number of faults that can be located when a certain percentage of code is examined by a developer. The x-axis is labeled with T-score. For LIBLIT05 and SOBER, we choose the top five predicates to form the set of blamed vertices. Because localization that still requires developers to examine more than 20 percent of the code is generally useless, we treat only [0, 20] as the meaningful T-score range. Under these circumstances, SOBER is apparently better than LIBLIT05, while both of them are consistently superior to CT.

For practical use, it is instructive to know how many (or what percentage of) faults can be identified when no more than α percent of the code is examined. We therefore plot the cumulative comparison in Fig. 4b. It clearly suggests that both SOBER and LIBLIT05 are much better than CT and that SOBER outperforms LIBLIT05 consistently. Although LIBLIT05 catches up when the T-score is 60 percent or higher, we regard this advantage as irrelevant because it hardly makes sense for a fault locator to require a developer to examine more than 60 percent of the code.

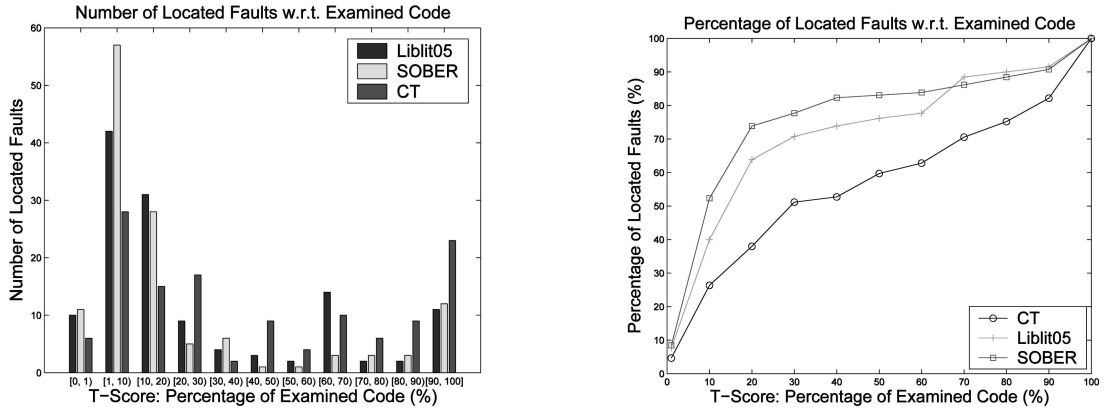


Fig. 4. Located faults with regard to code examination. (a) Interval comparison. (b) Cumulative comparison.

Fig. 4b shows that, for the 130 faults in the Siemens suite, when a developer examines at most 1 percent of the code, CT catches 4.65 percent of the faults while LIBLIT05 and SOBER capture 7.69 percent and 8.46 percent, respectively. Moreover, when 10 percent code examination is acceptable, CT and LIBLIT05 identify 34 (26.36 percent) and 52 (40.00 percent) of the 130 faults. SOBER is the best of the three, locating 68 (52.31 percent) of the 130 faults, which is 16 faults more than the state-of-the-art approach LIBLIT05. If the developer is patient enough to examine 20 percent of the code, 73.85 percent of the faults (i.e., 96 of 130) can be located by SOBER.

We also vary the parameter k in calculating the T-score for both LIBLIT05 and SOBER. The quality comparison is plotted in Fig. 5 for k varying from 1 through 8. The comparison is confined within the [0, 20] T-score range. Since detailed results about CT is not available in [6], CT is still depicted only at the 1, 10, and 20 ticks. Fig. 5 shows that LIBLIT05 is the best when k is equal to 1 or 2. When $k = 3$, SOBER catches up, and it consistently outperforms LIBLIT05 afterward. Because developers are always interested in locating faults with minimal code checking, it is desirable to select the optimal k that maximizes the localization quality. We found that both LIBLIT05 and SOBER achieve their best quality when k is equal to 5. In addition, Fig. 6 plots the quality of SOBER with various k-values. It clearly indicates that SOBER locates the largest number of faults when k is equal to 5. Therefore, the setting of $k = 5$ in Fig. 4 is justified. Finally, Fig. 6 also suggests that too few predicates (e.g., $k = 1$) may not convey enough information for fault localization, while too many predicates (e.g., $k = 9$) are in themselves a burden for developers to examine and, thus, neither of them leads to the best result.

Besides being accurate in fault localization, SOBER is also computationally efficient. Suppose we have n correct and m incorrect executions. Then, the time complexity of scoring

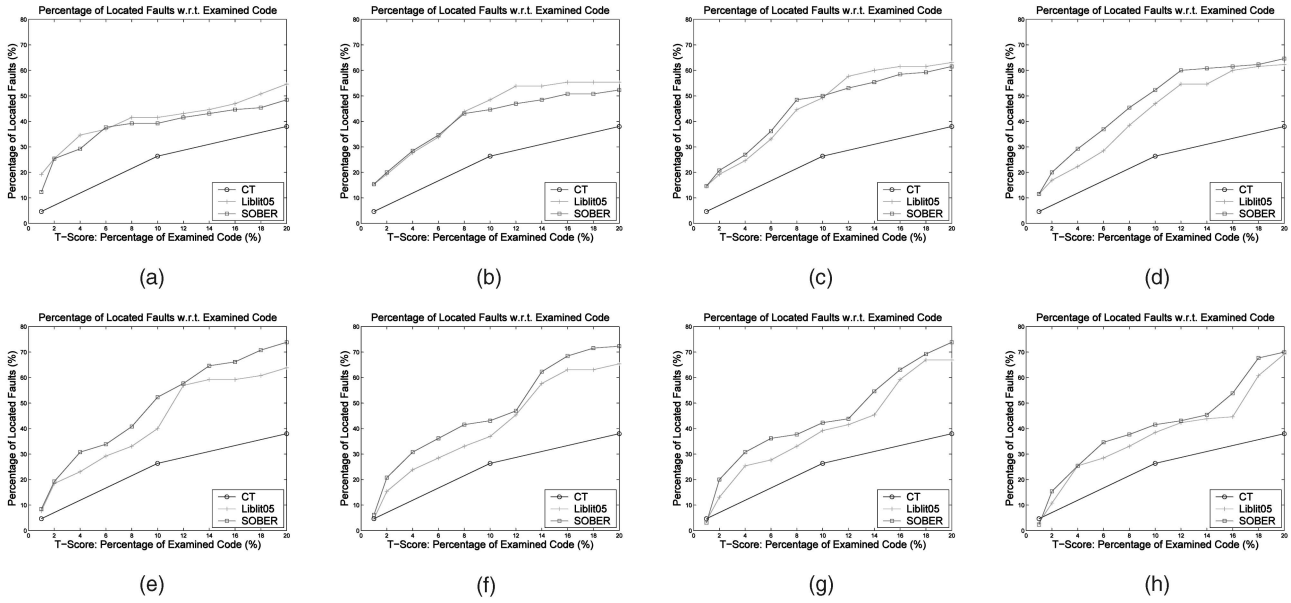


Fig. 5. Quality Comparison with regard to various top k Values. (a) Top one. (b) Top two. (c) Top three. (d) Top four. (e) Top five. (f) Top six. (g) Top seven. (h) Top eight.

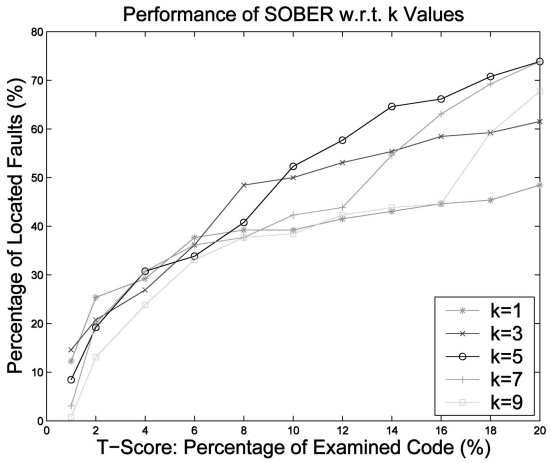


Fig. 6. Quality of SOBER with regard to top-k values.

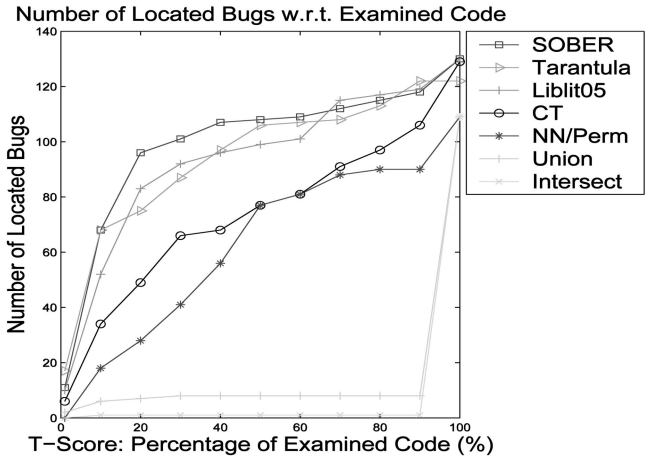


Fig. 7. Quality comparison between existing algorithms.

each predicate is $O(n+m)$. If there are, in total, k predicates instrumented, the entire time complexity of SOBER is $O((n+m)\cdot k + k\cdot log(k))$. Similarly, LIBLIT05 also needs $O(n+m)$ to score each predicate, and its time complexity is also $O((n+m)\cdot k + k\cdot log(k))$. We experimented with the 31 faulty versions of the replace program, and the average time for unoptimized LIBLIT05 and SOBER to analyze each version was 11.7775 seconds and 11.3844 seconds, respectively. This is much faster than CT, as reported in [6].

4.4 Comparison with TARANTULA and SLICECHOP

We now compare SOBER with TARANTULA and SLICECHOP. Recently, Jones and Harrold. [8] reported the result of TARANTULA on the Siemens suite with the ranking-based T-score, and compared it with previous PDG-based T-scores of CT, NN/PERM, INTERSECT, and UNION. As it is unclear to what extent these two kinds of T-score agree with each other, we assume they are equivalent, as Jones and Harrold did in [8]. More investigation, however, is needed to clarify this issue in the future. Moreover, because the authors failed to compare TARANTULA with statistical debugging in [8], this study fills the gap.

We differ from previous comparisons in choosing to compare algorithms in terms of the absolute number of faulty versions on which an algorithm renders a T-score of no more than α percent. Previously, different subsets of the Siemens suite were used by different authors, and the percentages based on the different subsets are put together for comparison [4], [6], [8], [9]. Specifically, the reported percentages for UNION, INTERSECT and NN/PERM are based on 109 faulty versions, and the percentage for CT is based on 129 versions. In the previous section, the percentages for LIBLIT05 and SOBER are calculated on the whole-set 130 versions. In a recent study of TARANTULA and SLICECHOP [8], [9], 122 and 38 faulty versions are used by the original authors, respectively.

Therefore, based on the reported percentage and the chosen subset of faulty versions, we recover how many faults are located by each algorithm with a T-score no more than α percent, and Fig. 7 shows the effectiveness comparison in terms of the absolute number of faulty versions. Because the study of SLICECHOP *excluded* 91 faulty versions, for fairness it is not plotted in Fig. 7. Instead, we compare SOBER with SLICECHOP separately later.

Fig. 7 clearly shows that the effectiveness of the seven algorithms is at three different levels. The algorithms UNION and INTERSECT are the least effective and NN/PERM and CT are in the middle, with CT being better than NN/PERM. The other three algorithms: LIBLIT05, TARANTULA and SOBER, apparently have the best result on the Siemens suite.

We now compare TARANTULA with SOBER in detail. They both locate 68 faults when the T-score is no more than 10 percent. When the T-score is less than 1 percent, TARANTULA and SOBER locate 17 and 11 faults, respectively. On the other hand, with the T-score no more than 20 percent, SOBER can help locate 96 out of the 130 faults, whereas TARANTULA helps locate 75. Since the comparison is based on the assumption of the equivalence between the PDG-based and ranking-based T-scores, we refrain from drawing conclusions about the relative superiority of either method. Ultimately, the effectiveness of all fault localization algorithms will be assessed by end-users in practice.

We now compare SOBER with SLICECHOP. In the study of SLICECHOP [9], the authors excluded the program tcas from the Siemens suite due to its small size, and they exclude the program tot_info because, at that time, their framework could not handle floating point operations. For the remaining five subject programs, which consist of 66 faulty versions in total, another 28 faulty versions were excluded for various reasons, leaving 38 versions used in the final evaluation. The authors reported that for 23 out of the 38 versions, no more than 10.4 percent of the source code needed to be examined. We checked the quality of SOBER on the 66 versions of the five subject programs, and found that the T-score is less than 10.4 percent on 43 versions. Moreover, within the 38 faulty versions examined by SLICECHOP in [9], SOBER has a T-score of less than 8.4 percent on 27 versions. Because the ratio of examined code was not reported for each of the 38 versions in [9], no further comparison is performed here between SOBER and SLICECHOP.

4.5 Comparison from Other Perspectives

A comprehensive comparison between fault localization algorithms is hard, and many aspects must be considered for a fair comparison. For example, some important aspects are the runtime overhead, analysis complexity, localization accuracy, and the accessibility of final fault localization reports. So far, we have been focusing on localization accuracy and have demonstrated that SOBER is one of the most accurate algorithms. However, when compared on other aspects, SOBER might be inferior to other techniques, at least in its current state.

First, some techniques, like NN/PERM, CT and SLICECHOP, only need one failing and multiple passing cases for fault localization, whereas SOBER, LIBLIT05 and TARANTULA, in principle, need to collect the statistics from multiple failing cases. Secondly, SOBER could be inferior to LIBLIT05 in terms of the runtime overhead due to instrumentation. Specifically, since LIBLIT05 is based on the predicate coverage data, the instrumentation on a predicate can be disabled once the predicate has been evaluated (in a similar way to Jazz [25]). In contrast, SOBER needs to count the evaluation frequency throughout the execution. Finally, some algorithms, including TARANTULA, LIBLIT05 and Delta Debugging, have provided visual interfaces to increase their accessibility. Currently, no visual interface is available for SOBER, but one could be added in the future.

5 SOBER IN AN IMPERFECT WORLD

Besides the probabilistic treatment of program predicates, there are two other factors that implicitly contribute to SOBER's effectiveness shown in Section 4. First, the test suite in the experiment is reasonably adequate given the program code size: Each subject program of the Siemens suite is accompanied by a few thousand test cases.[1] Intuitively, more-reliable statistics can be collected from a more-adequate test suite and would enable SOBER to produce better fault localizations. Second, by taking the fault-free version as the test oracle, each execution is precisely labeled as either passing or failing. This provides SOBER with a noise-free analysis environment, which likely benefits SOBER's inference ability.

Although these two elements are highly desirable for quality localization, they are usually not available in practice due to the potential high cost. For example, because the program specification varies from one component to another, exclusive test scripts for each component must be prepared by human testers. Although some tools can help expedite the generation of test cases [26], [27], [28], critical manual work is still unavoidable. Furthermore, besides the difficulty of test case generation, the test oracle is even harder to construct. Again because of variations in program functionality, it is usually humans developers who prepare the expected outputs or pass judgment about the correctness of outputs in practice.

1. In this paper, we take the number of test cases as a rough measure of the test adequacy. More involved discussion about test adequacy is out of the scope of this study.

Therefore, considering the difficulty of obtaining an adequate test suite and a test oracle, we regard the environment that we experimented with in Section 4 as "a perfect world." In order to shed some light on how SOBER would work in practice, in this section we subject SOBER to an "imperfect world," where adequate test suites and test oracles are not simultaneously available. Section 5.1 examines SOBER's robustness to test inadequacy, and Section 5.2 studies how SOBER handles partially labeled test suites.

We regard, and hence believe, that the examination of SOBER in an "imperfect world" is both necessary and interesting. To some extent, this examination bridges the gap between the perfect-world experiments (i.e., Section 4) and real-world practices that cannot be fully covered in any single research paper. We simulate the imperfect world with the 130 faulty versions of the Siemens suite. In parallel with SOBER, LIBLIT05 is also subjected to the same experiments for a comparative study, which illustrates how the two statistical debugging algorithms react to the imperfect world.

5.1 Robustness to Inadequate Test Suites

Because of the cost of an adequate test suite, people usually settle for inadequate but nevertheless satisfactory suites in practice. For instance, during the prototyping stage, one may not bother much with an all-around testing, and a preliminary test suite usually suffices. We now simulate an inadequate test suite by sampling (without replacement) the accompanying test suite of the Siemens suite. The sampled test suite becomes more and more inadequate as the sampling rate gets smaller.

Specifically, for each faulty version of the Siemens suite, we randomly sample a portion β $(0 < \beta \leq 1)$ of the original test suite T. Suppose T consists of N test cases. Then, $\lceil N * \beta \rceil$ cases are randomly taken, constituting a β-sampled test suite, denoted as T_β. Because both SOBER and LIBLIT05 need at least one failing case, the above sampling is repeated until at least one failing case is included. Finally, both SOBER and LIBLIT05 are run on the *same* T_β for each faulty program.

Fig. 8 plots how the quality varies with different sampling rates for both SOBER and LIBLIT05. We set β equal to 100 percent, 10 percent, 1 percent and 0.1 percent, respectively, so that $T_{100\%}$ represents the entire test suite and each of the following is roughly one-tenth as small as the previous one. As β gets smaller, the localization quality of both SOBER and LIBLIT05 gradually degrades. For example, in Fig. 8a, curves for smaller βs are *strictly* below those for higher sampling rates. A similar pattern for LIBLIT05 is also observed in Fig. 8b. These observations are easily explainable. In statistical hypothesis testing, the confidence of either *accepting* or *rejecting* the null hypothesis is, in general, proportional to the number of observations. Because SOBER bears a similar rationale to hypothesis testing, its quality naturally improves as more and more test cases are observed. Because LIBLIT05 relies on the accurate estimation of the two conditional probabilities, its quality also improves with more labeled test cases due to the Law of Large Numbers.

In Fig. 8a, one can also notice that the curve for $\beta = 10\%$ is quite close to the highest. This suggests that SOBER

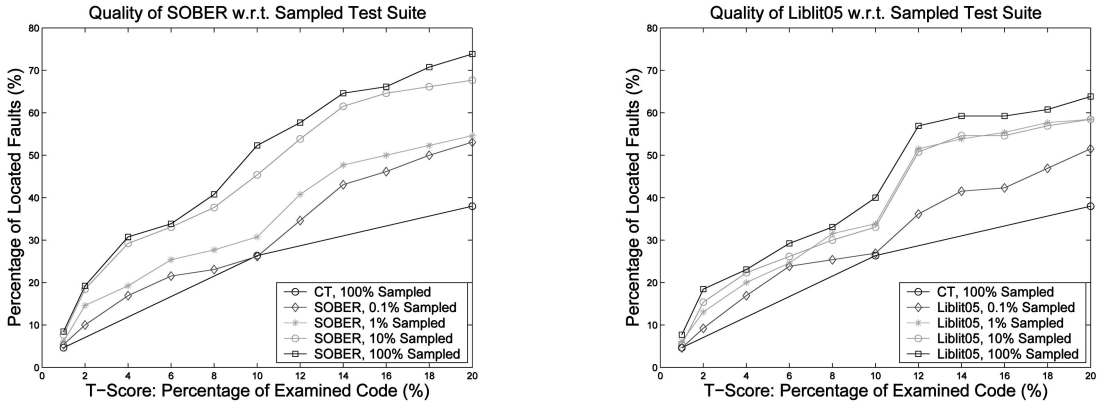


Fig. 8. Quality degradation with regard to β percent-sampled test suite. (a) Quality of SOBER with regard to sampled test suite. (b) Quality of LIBLIT with regard to sampled test suite.

obtains competitive results even when the test suite is only one-tenth of the original. Moreover, Fig. 8 also indicates that even when β is as low as 0.1 percent, both SOBER and LIBLIT05 are still consistently better than CT. Based on the typical suite size from Table 1, $T_{0.1\%}$ contains at most six test cases, at least one of which is failing. As one can see, even with such an insufficient test suite, both SOBER and LIBLIT05 still outperform CT. For example, without examining more than 20 percent of the code, SOBER and LIBLIT05 locate 53.08 percent and 51.54 percent of the 130 faults respectively, while CT works well with 38 percent of the versions. This could be attributed to the underlying mechanism of CT: It localizes faults by systematically contrasting the memory graphs of one passing and one failing execution. However, because the faults in the Siemens suite are mainly logic errors that rarely cause memory abnormalities, CT has difficulties in identifying the "delta" and further locating the fault. On the other hand, because predicates express logic relations, it is no surprise that predicate-based algorithms work better.

Beside varying the sampling rate β, we also examined how the quality changed with respect to the *absolute* size of the test suite. However, because the size of the accompanying test suite and the failing rate drastically vary from one faulty version to another, it makes little sense to set a uniform size for the test suite for quality examination. We therefore refrain from doing so, but choose instead to study how the number of failing cases could affect the localization quality, as described in the next section.

5.2 Handling Partially Labeled Test Suites

Although an adequate test suite is difficult to obtain, preparing a test oracle that can automatically recognize each execution as either passing or failing is even harder. In some situations, test case generation can be relatively easy. For example, one can simply feed random strings to a program that consumes string inputs. However, these test cases are hardly useful until we know the expected outputs.

In practice, except for programs that can be described by a program model, the expected outputs are usually prepared by human testers, either manually or assisted by tools. It is usually unrealistic for a tester to examine thousands of executions and label them. Instead, a tester will likely stop testing and return the faulty program to developers for patches when a small number of failing cases are encountered. At that time, the examined cases are *labeled* and the rest are *unlabeled*. This describes a typical scenario which exemplifies how partially labeled test suites arise in practice. In this section, we examine how well SOBER helps developers locate the underlying faults, when the test suite is partially labeled.

Formally, given a test suite T, suppose a tester has examined and labeled a subset suite T_e ($T_e \subseteq T$). Because manual labeling is usually expensive, it is common that $|T_e| \ll |T|$. Let T_p and T_f denote the set of passing and failing runs identified by the tester. Then, $T_e = T_p \cup T_f$ and $T_p \cap T_f = \phi$. We use T_u to denote the unexamined part of the suite, i.e., $T_u = T - T_e$. T is partially labeled if and only if $T_u \neq \phi$. The set relationship is further depicted in Fig. 9a. The outer ellipse represents the entire test suite T. The vertical line divides T into the full failing set T_f^t on the left and the full passing set T_p^t on the right. Certainly, $T_f \subseteq T_f^t$ and $T_p \subseteq T_p^t$. As seen in Fig. 8a, the best localization is achieved by SOBER when $T_f = T_f^t$ and $T_p = T_p^t$, i.e., when the given test suite T is fully labeled.

Now, given the partially labeled test suite T, the most straightforward scheme for SOBER is to analyze labeled test cases T_e only. Because T_e is fully labeled, SOBER can be immediately applied with T_e. In fact, this scheme is

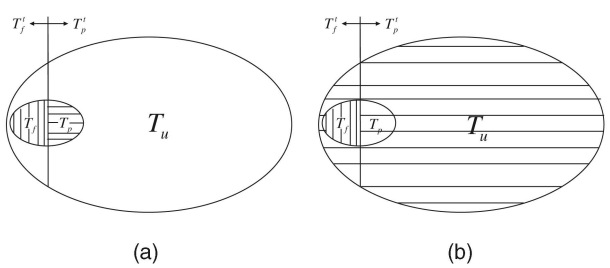


Fig. 9. Two schemes to work with partial-labeled test suite. (a) Scheme with labeled cases only. (b) Scheme with both labeled and unlabeled cases.

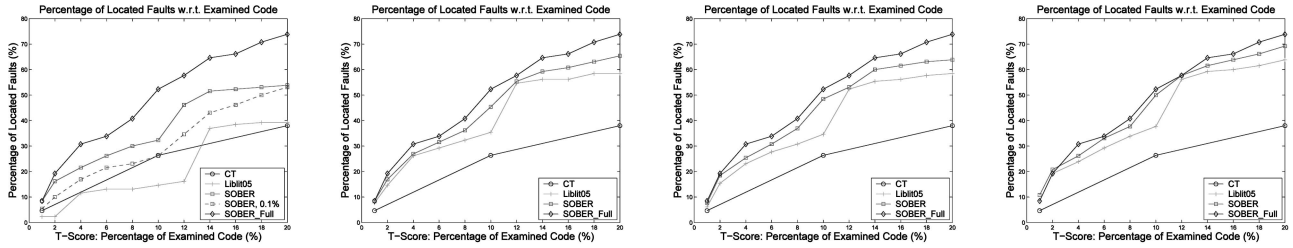


Fig. 10. Quality comparison with regard to the number (m) of labeled failing cases. (a) $m = 1$. (b) $m = 3$. (c) $m = 5$. (d) $m = 10$.

equivalent to running SOBER on a β-sampled test suite, where $\beta \approx \frac{|T_e|}{|T|}$ and is usually quite small. As a conservative estimation, β can be around 1 percent. In considering the Siemens programs, $\beta = 1\%$ means that the tester examines tens among thousands of test cases and identifies about five failing runs on the average. In our opinion, this can be a reasonable workload for the tester.

This scheme, although straightforward, does render reasonable localization results. As shown in Fig. 8a, "SOBER 1% Sampled" is clearly better than CT. But it is also seen that a considerable gap exists between "SOBER 0.1% Sampled" and "SOBER 100% Sampled." For concise reference, we use SOBER_FULL to denote "SOBER 100% Sampled" in the following. Although the same quality as SOBER_FULL is not (unrealistically) expected when T is partially labeled, we nevertheless believe that T_u can be utilized for better quality than that with T_e only.

The above straightforward scheme apparently overlooks the information contained in T_u. Although T_u bears no labeling information, its runtime statistics, if used properly, can assist SOBER in fault analysis. In this study, we restrict our discussion to reasonably developed programs that pass all but a few test cases. One can judge whether this assumption holds by examining the percentage of failing cases in T_e. For example, if a program fails most cases in T_e, the fault could be quite easy to find. For reasonably developed programs, we can choose to label all the unexamined test cases T_u as passing and apply SOBER to the *regarded* failing and passing set T'_f and T'_p, where $T'_f = T_f$ and $T'_p = T_p \cup T_u$. The difference between the two schemes is visualized in Fig. 9.

Let T_m represent the set of unexamined failing cases, i.e., $T_m = T^t_f - T_f$. Then, all the cases in T_m are mislabeled as passing in the above treatment. While this mislabeling unavoidably introduces impurity into T'_p, the effect it has on SOBER is minimal: The θ_p calculated with T'_p deviates negligibly from that with T^t_p because $T'_p = T_p \cup T_u = T^t_p \cup T_m$ and $|T_m| \leq |T_f| \ll |T^t_p|$.

On the other hand, by mislabeling T_m, we utilize the runtime statistics of the cases in $T^t_p - T_p$, which are otherwise disregarded. In this way, θ_p can be estimated more accurately with T'_p than with T_p only. This could subsequently bring better localization quality. Therefore, this is essentially a trade-off between grabbing more passing runs and (unavoidably) mislabeling some failing runs. In our belief, the gain from including more passing executions should surpass the loss from mislabeling. As will be shown shortly, this scheme achieves much better results than the straightforward scheme, and sometimes it even obtains comparable results to SOBER_FULL.

We simulate partially labeled test suites using the Siemens programs. For each faulty version, we randomly select m failing cases as T_f (i.e., the set of failing cases identified by the tester). According to the above scheme, all the remaining cases are regarded as passing, i.e., T'_p. We then run both SOBER and LIBLIT05 with the same T'_p and T'_f (recall that $T'_f = T_f$) for each of the 130 faulty versions. We experiment with m equal to 1, 3, 5, and 10, respectively, and this represents the increasing effort that the tester puts into test evaluation. If a faulty version does not have m failing cases, we take all the failing cases. In the Siemens suite, there are 0, 4, 14, and 19 versions that have less than 1, 3, 5, and 10 failing cases. These versions were not excluded because they do represent real situations.

Fig. 10 plots the localization quality for both SOBER and LIBLIT05 with m equal to 1, 3, 5, and 10, respectively. Curves for CT and SOBER_FULL are also plotted as the baseline and ceiling quality in each subfigure. Among the four subfigures, Fig. 10a represents the toughest situation, where only one failing case is identified in each faulty version. This simulates a typical scenario where a developer starts debugging once a faulty execution is encountered. As expected, the quality of SOBER degrades considerably from SOBER_FULL, but it is still better than CT.

We note that the $m = 1$ situation is at least as harsh as the situation with 0.1 percent-sampled test suites, as shown in Fig. 8a. Nevertheless, at least one failing run is in every 0.1 percent-sampled test suite. In order to demonstrate the effect of treating T_u as passing, we replot the curve of SOBER with $\beta = 0.1\%$ in Fig. 10a with a dashed line. The remarkable gap between "SOBER" and "SOBER, 0.1%" suggests the benefit of treating unlabeled cases as passing.

The four subfigures of Fig. 10, viewed in a sequence, show that the quality of SOBER gradually improves as additional failing cases are explicitly labeled. Intuitively, the more failing cases that are identified, the more accurately the statistic Y (4) approaches to the true faulty behavior of predicate P and, hence, the higher quality of the final predicate ranking list. LIBLIT05 also improves for a similar reason.

5.3 Summary

In this section, we empirically examined how SOBER works in an imperfect world, where either the test suite is

```
    static int grep(int fd)
    {
     ...
541  for( ; ; )
542  {
     ...
548   lastnl = bufbeg;
549   if (lastout)                    P1
550       lastout = bufbeg;
551   if (buflim - bufbeg == save)
552     break;
553   beg = bufbeg + save - residue + 1; /* fault 1 */
554   for(lim = buflim; lim > beg && lim[-1] != '\n'; --lim)
555     ;
      ...
566   beg = lim;
567   while(i < out_before && beg > bufbeg && beg != lastout)
568   {
569      ++i;
570      do
571        --beg;
572      while(beg > bufbeg && beg[-1] != '\n');
573   }
574   if (beg != lastout)                 P2
57 5    lastout = 0;
576   save = residue + lim - beg;
      ...
580  }
      ...
587  return nlines;
588 }
```

Fig. 11. Fault 1: An off-by-one error in grep.c.

```
     static char ** comsubs(char* left, char* right)
     {
       ...
2264   for(lcp = left; *lcp != '\0'; ++lcp)
2265   {
2266     len = 0;
2267     rcp = index(right, *lcp);
2268     while (rcp != NULL)                          P3
2269     {
2270       for (i = 1; lcp[i] != '\0'
           /* && lcp[i] == rcp[i] */; ++i) /* fault 2 */
2271         continue;
2272       if (i > len)
2273         len = i;
2274       rcp = index(rcp + 1, *lcp);
2275     }
2276     if (len == 0)
2277       continue;
2278     if ((cpp = enlist(cpp, lcp, len)) == NULL)
2279       break;
2280   }
2281   return cpp;
2282 }
```

Fig. 12. Fault 2: A subclause-missing error in dfa.c.

inadequate or only a limited number of failing executions are explicitly identified. The experiment demonstrates the robustness of SOBER under these harsh conditions. In addition, the scheme of tagging all unlabeled cases as passing is shown effective in leveraging SOBER's quality.

6 Experimental Evaluation with Large Programs

Although the 130 faulty versions of the Siemens programs are appropriate for algorithm comparison, the effectiveness of SOBER nevertheless needs to be assessed on large programs. In this section, we report on the experimental evaluation of SOBER on two (reasonably) large programs, grep 2.2 and bc 1.06. Moreover, as two faults are located in each program, this evaluation also illustrates how SOBER helps developers handle multifault cases. The detailed experimental results with grep 2.2 and bc 1.06 are presented in Sections 6.1 and 6.2, respectively.

6.1 A Controlled Experiment with grep 2.2

We obtained a copy of the grep 2.2 subject program from the "Subject Infrastructure Repository" (SIR) [29]. The original code of grep 2.2 has 11,826 lines of C code, as counted by the tool SLOCCount [30], while the announced size of the modified version at SIR is 15,633 LOC. A test suite of 470 test cases is available at SIR for the program. We tried out all the seeded faults provided by SIR, but found no fault incurred failures on the accompanying test suite. We therefore manually injected two faults in the source code, as shown in Fig. 11 and Fig. 12, respectively.

The first fault (shown in Fig. 11) is an "off-by-one" error: an expression "+1" is appended to line 553 in the grep.c file. This fault causes failures in 48 of the 470 test cases. The second fault (in Fig. 12) is a "subclause-missing" error. The subclause (`lcp[i] == rcp[i]`) is commented out at line 2270 in file dfa.c. The fault incurs another 88 failing cases.

Although these two faults are manually injected, they do mimic realistic logic errors. Logic errors like "off-by-one" or "subclause-missing" may sneak in when developers are handling obscure corner conditions. Because logic errors, like these two, do not generally incur program crashes, they are usually harder to debug than those causing program crashes. In the following, we illustrate how SOBER helps developers find these two faults.

We first instrument the source code. According to the instrumentation schema described in Section 4.1, grep 2.2 is instrumented with 1,732 branch and 1,404 return predicates. The first run of SOBER with the 136 failing (due to the two faults) and the remaining 334 passing cases produces a predicate ranking, whose top three predicates are listed in Table 2. For easy reference, the three predicates are also marked at their instrumented locations in Fig. 11 and Fig. 12.

As we can see, the predicates P_1 and P_2 point to the faulty function for the first fault. The predicate P_1 is four lines above the real fault location. The predicate P_3, on the other hand, points directly to the *exact* location of the second fault. Now, let us explore how these top predicates help developers locate the faults.

Given the top-ranked predicates, it is natural to ask why they are ranked high. We find that the sample mean and standard deviation of the evaluation bias of P_1 (denoted by $\pi(P_1)$) are 0.90 and 0.25 in the 136 failing cases but are 0.99

TABLE 2
Top Three Predicates from the First Run of SOBER

Ranks	Filename	Line Num.	Predicate
P_1	grep.c	549	(lastout) == true
P_2	grep.c	574	(beg != lastout) == true
P_3	dfa.c	2270	(lcp[i] != '\0') == true

```
    void more_variables ()
    {
      ...
127   old_count = v_count;
      ...
137   for (indx = 3; indx < old_count; indx++)
      ...
141   for (; indx < v_count; indx++)
      ...
    }
```

Fig. 13. First Fault in bc 1.06, in storage.c.

```
    void more_arrays ()
    {
      ...
167   arrays = (bc_var_array **) bc_malloc (a_count
              * sizeof(bc_var_array*));
      ...
      /* Copy the old arrays. */
      for (indx = 1; indx < old_count; indx++)
        arrays[indx] = old_ary[indx];

176   for (; indx < v_count; indx++)
        arrays[indx] = NULL;
      ...
    }
```

Fig. 14. Second fault in bc 1.06, in storage.c.

and 0.065 in the remaining 344 passing cases. This suggests that P_1 is mostly evaluated `true` in passing cases with a small variance but is mostly evaluated `false` in *some* failing cases, as indicated by its much larger variance in failing cases. By examining $\pi(P_1)$ in the failing cases, we find that $\pi(P_1)$ is smaller than 0.1 in five failing cases. Therefore, we know that P_1 can be evaluated mostly as `false` in these failing cases, whereas it is mostly `true` in passing cases. Similarly, we find that P_2 is considerably evaluated as `true` in failing cases, but mostly `false` in passing cases.

We notice that when P_2 evaluates `true`, the variable `lastout` is reset to 0, which immediately causes P_1 to evaluate as `false` in the next iteration. This explains why predicates P_1 and P_2 are both ranked at the top. In order to find why "`beg != lastout`" tends to evaluate to `true` in failing cases, a developer would pay attention to the assignment to variables `beg` and `lastout`. Within the `for` loop from lines 541 through 580, there are no other assignments to `lastout` except lines 549 and 575. Then, the developer would examine lines 553, 566, and 571, where `beg` gets assigned. A developer familiar with the code will then identify the fault.

After fixing the first fault, a second run of SOBER with the 88 failing and 382 passing cases puts P_3 at the top. A developer paying more attention to line 2270 of the dfa.c file would find the fault, as P_3 points to the exact fault location. Because SOBER is only for fault localization, it is the developer's responsibility to confirm the fault location and fix it. To the best of our knowledge, no tools can automatically suggest patches for logic errors without assuming any specifications.

6.2 A Case Study with bc 1.06

In this section, we report a case study of SOBER with a real-world program bc 1.06, on which SOBER identifies two buffer overflow faults, one of which has never been reported before.

bc is a calculator program that accepts scripts written in the bc language, which supports arbitrary precision calculations. The 1.06 version of the bc program is shipped with most recent UNIX/Linux distributions. It has 14,288 LOC, and a buffer overflow fault has been reported in [7], [12].

This experiment was conducted on a 3.06 GHz Pentium-4 PC running Linux RedHat 9 with gcc 3.3.3. Inputs to bc 1.06 are 4,000 valid bc programs that are randomly generated with various size and complexity. We generate each input program in two steps: First, a random syntax tree is generated in compliance with the bc language specification; second, a program is derived from the syntax tree.

With the aid of SOBER, we quickly identify two faults in bc 1.06, including one that has not been reported. Among the 4,000 input cases , the bc 1.06 program fails 521 of them. After running through these test cases, the analysis from SOBER reports "`indx < old_count`" as the most fault-relevant predicate. This predicate points to the variable `old_ count` in line 137 of storage.c (shown in Fig. 13). A quick scan of the code shows that `old_count` copies its value from `v_count`. By putting a watch on `v_count`, we find that `v_count` is overwritten when a buffer named `genstr` overflows (in bc.y, line 306). The buffer `genstr` is 80 bytes long and is used to hold bytecode characters. An input containing complex and relatively large functions can easily overflow it. To the best of our knowledge, this fault has not been reported before. We manually examine the statistics of the top-ranked predicate and find that its evaluation bias in correct and incorrect executions is 0.0274 and 0.9423, respectively, which intuitively explains why SOBER works. LIBLIT05 also ranks the same predicate at the top.

After fixing the above fault, a second run of SOBER (3,303 correct and 697 incorrect cases) generates a fault report with the top predicate "`a_count < v_count`," which points to line 176 of storage.c (shown in Fig. 14). This is likely a copy-paste error where `a_count` should have been used in the position of `v_count`. This fault has been reported in previous studies [7], [12].

As a final note, predicates identified by SOBER for these two faults are far from the actual crashing points. This suggests that SOBER picks up predicates that characterize the scenario under which faults are triggered, rather than the crashing venues.

7 Discussion

7.1 Related Work

In this section, we briefly review previous work related to fault detection in general. Static analysis techniques have been used to verify program correctness against a well-specified program model [1], [31] and to check real codes directly for Java [2] and C/C++ programs [3]. Engler et al. [32] further show that the correctness rules sometimes can be automatically inferred from source code, hence saving, to some extent, the cost of preparing specifications. Complementary to static analysis, dynamic analysis focuses more

on the runtime behavior and often assumes fewer specifications. SOBER belongs to the category of dynamic analysis.

Within dynamic analysis, most fault localization techniques are based on the contrast between failing and passing cases [4], [5], [6], [7], [8], [12], [20], [21], [33]. For example, invariants that are formed from passing cases can suggest potential fault locations if they are violated in any failing cases [20]. Readers interested in the details of invariants are referred to the project DAIKON [16]. The DIDUCE project [14] monitors a more restricted set of predicates and relaxes them in a similar manner to DAIKON at runtime. After the set of predicates becomes stable, the DIDUCE tool relates future violations as indications of potential faults. This approach is demonstrated to be effective on four large software systems. However, as invariants are a special kind of predicates that hold in all passing executions, they may not be effective in locating subtle faults as suggested by Pytlik et al. in [20]. In comparison, the probabilistic treatment of predicates implemented by SOBER naturally relaxes this requirement and is shown to achieve much better localization results on the Siemens suite.

Contrasts based on program slicing [34] and dicing [35] are also shown effective for fault localization. For example, Agrawal et al. [33] present a fault localization technique, implemented as χslice, which is based on the execution traces of test cases. This technique displays and contrasts the dices of one failing case to those of multiple passing cases. Jones et al. [23] describe a similar approach implemented as TARANTULA. Unlike χ-slice, TARANTULA collects the testing information from all passing and failing cases and colors suspicious statements based on the contrast. Later, Renieris and Reiss [4] find that the contrast renders better fault localization when the given failing case is contrasted with the most similar passing case (i.e., the nearest neighbor). In comparison, SOBER collects the evaluation frequency of instrumented predicates, a much richer information base, and quantifies the model difference through a statistical approach.

While all the fault localization algorithms examined in this paper are designed for programming professionals, recent years have also witnessed an emergence of fault localization algorithms especially tuned to assist end users in fault diagnosis. For example, Ayalew and Mittermeir propose a technique to trace faults in spreadsheets based on "interval testing" and slicing [36]. Ruthruff et al. improve this approach by allowing end-users to interactively adjust their feedbacks [37]. The *Whyline* prototype realizes a new debugging paradigm called "interrogative debugging," which allows users to ask *why did* and *why didn't* questions about runtime failures [38].

The power of statistical analysis is demonstrated in program analysis and fault detection. Dickinson et al. find program failures through clustering program execution profiles [39]. Their subsequent work [40] first performs feature selection using logistic regression and then clusters failure reports within the space of selected features. The clustering results are shown to be useful in prioritizing software faults. Early work of Liblit et al. on statistical debugging [7] also adopts logistic regression in sifting predicates that are correlated with program crashes. In addition, they impose $\mathcal{L}_1$ norm regularization during the regression so that predicates that are really correlated are distinguished. In comparison, our method SOBER is a statistical model-based approach, while the above statistical methods follow the principle of discriminant analysis. Specifically, SOBER features a hypothesis testing-based approach, which has not been seen in the fault localization literature.

7.2 Threats to Validity

Like any empirical study, threats to validity should be considered in interpreting the experimental results presented in this paper. Specifically, the results obtained with the Siemens suite cannot be generalized to arbitrary programs. However, we expect that on larger programs with greater separation of concerns, most fault localization techniques will do better. This expectation is supported by existing studies with CT, LIBLIT05, and TARANTULA [6], [8], [12], as well as the experiments in Section 6 in this study.

Threats to construct validity concern the appropriateness of the quality metric for fault localization results. In this paper, we adopt the PDG-based T-score, which was proposed by Renieris and Reiss [4]. Although this evaluation framework involves no subjective judgments, it is by no means a comprehensively fair metric. For instance, this measure does not take into account how easily a developer can make sense of the fault localization report. Recent work [6] also identifies some other limitations of this measurement. In previous work, a ranking-based T-score is used to evaluate the effectiveness of TARANTULA. Although both forms of T-score estimate the human efforts needed to locate the fault, it is yet unclear whether they agree. The comparison of TARANTULA with other algorithms in Section 4.4 assumes the equivalence between the two forms. More extensive studies are needed to clarify this issue.

Finally, threats to internal validity concern the experiments of SOBER with the programs grep 2.2 and bc 1.06, discussed in Section 6. Specifically, the two *logic errors* in grep 2.2 are injected by us. However, because these two logic errors do not incur segmentation faults, they are generally harder to debug, even for human developers. In contrast, case studies in previous work target crashing faults [5], [6], [7], [12]. Therefore, the experiment with grep 2.2 demonstrates the effectiveness of SOBER on large programs with logic errors. In order to minimize the threats to external validity about experiments with large programs, a case study with bc 1.06 is also presented, which illustrates the effectiveness of SOBER on real faults. However, two experiments are still insufficient to make claims about the general effectiveness of SOBER on large programs. Ultimately, all fault localization algorithms should be subjected to real practice, and evaluated by end users.

8 Conclusions

In this paper, we propose a statistical approach to localize software faults without prior knowledge of program semantics. This approach tackles the limitations of previous methods in modeling the divergence of predicate evaluations between correct and incorrect executions. A systematic

evaluation with the Siemens suite, together with two case studies with grep 2.2 and bc 1.06, clearly demonstrates the advantages of our method in fault localization. We also simulate an "imperfect world" to investigate SOBER's robustness to the harsh scenarios that may be encountered in practice. The experimental result favorably supports SOBER's applicability.

ACKNOWLEDGMENTS

The authors would like to thank Gregg Rothermel for making the Siemens program suite available. Darko Marinov provided the authors with insightful suggestions. Andreas Zeller, Holger Cleve and Manos Reneris generously shared their evaluation frameworks. GrammaTech Inc. offered the authors a free copy of CODESURFER. Last but not the least, the authors deeply appreciate the insightful questions, comments, and suggestions from anonymous referees, which proved invaluable during the preparation of this paper.

REFERENCES

[1] E. Clarke, O. Grumberg, and D. Peled, *Model Checking*. MIT Press, 1999.

[2] W. Visser, K. Havelund, G. Brat, and S. Park, "Model Checking Programs," *Proc. 15th IEEE Int'l Conf. Automated Software Eng. (ASE '00)*, pp. 3-12, 2000.

[3] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill, "CMC: A Pragmatic Approach to Model Checking Real Code," *Proc. Fifth Symp. Operating System Design and Implementation (OSDI '02)*, pp. 75-88, 2002.

[4] M. Renieris and S. Reiss, "Fault Localization with Nearest Neighbor Queries," *Proc. 18th IEEE Int'l Conf. Automated Software Eng. (ASE '03)*, pp. 30-39, 2003.

[5] A. Zeller, "Isolating Cause-Effect Chains from Computer Programs," *Proc. ACM Int'l Symp. Foundations of Software Eng. (FSE '02)*, pp. 1-10, 2002.

[6] H. Cleve and A. Zeller, "Locating Causes of Program Failures," *Proc. 27th Int'l Conf. Software Eng. (ICSE '05)*, pp. 342-351, 2005.

[7] B. Liblit, A. Aiken, A. Zheng, and M. Jordan, "Bug Isolation via Remote Program Sampling," *Proc. ACM SIGPLAN 2003 Int'l Conf. Programming Language Design and Implementation (PLDI '03)*, pp. 141-154, 2003.

[8] J. Jones and M. Harrold, "Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique," *Proc. 20th IEEE/ACM Int'l Conf. Automated Software Eng. (ASE '05)*, pp. 273-282, 2005.

[9] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating Faulty Code Using Failure-Inducing Chops," *Proc. 20th IEEE/ACM Int'l Conf. Automated Software Eng. (ASE '05)*, pp. 263-272, 2005.

[10] I. Vessey, "Expertise in Debugging Computer Programs," *Int'l J. Man-Machine Studies: A Process Analysis*, vol. 23, no. 5, pp. 459-494, 1985.

[11] M. 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*, vol. 10, no. 3, pp. 171-194, 2000.

[12] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan, "Scalable Statistical Bug Isolation," *Proc. ACM SIGPLAN 2005 Int'l Conf. Programming Language Design and Implementation (PLDI '05)*, pp. 15-26, 2005.

[13] Y. Brun and M. Ernst, "Finding Latent Code Errors via Machine Learning over Program Executions," *Proc. 26th Int'l Conf. Software Eng. (ICSE '04)*, pp. 480-490, 2004.

[14] S. Hangal and M. Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection," *Proc. 24th Int. Conf. Software Eng. (ICSE '02)*, pp. 291-301, 2002.

[15] G. Casella and R. Berger, *Statistical Inference*, second ed., Duxbury, 2001.

[16] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically Discovering Likely Program Invariants to Support Program Evolution," *IEEE Trans. Software Eng.*, vol. 27, no. 2, pp. 1-25, Feb. 2001.

[17] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria," *Proc. 16th Int'l Conf. Software Eng. (ICSE'94)*, pp. 191-200, 1994.

[18] G. Rothermel and M. Harrold, "Empirical Studies of a Safe Regression Test Selection Technique," *IEEE Trans. Software Eng.*, vol. 24, no. 6, pp. 401-419, June 1998.

[19] T. Cover and J. Thomas, *Elements of Information Theory*, first ed. Wiley-Interscience, 1991.

[20] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. Reiss, "Automated Fault Localization Using Potential Invariants," *Proc. Fifth Int'l Workshop Automated and Algorithmic Debugging (AADEBUG '03)*, pp. 273-276, 2003.

[21] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff, "Sober: Statistical Model-Based Bug Localization," *Proc. 10th European Software Eng. Conf./13th ACM SIGSOFT Int'l Symp. Foundations of Software Eng. (ESEC/FSE '05)*, pp. 286-295, 2005.

[22] T. Zimmermann and A. Zeller, "Visualizing Memory Graphs," *Revised Lectures on Software Visualization, Int'l Seminar*, pp. 191-204, 2002.

[23] J. Jones, M. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," *Proc. 24th Int'l Conf. Software Eng. (ICSE '02)*, pp. 467-477, 2002.

[24] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-Inducing Input," *IEEE Trans. Software Eng.*, vol. 28, no. 2, pp. 183-200, Feb. 2002.

[25] J. Misurda, J. Clause, J. Reed, B. Childers, and M. Soffa, "Jazz: A Tool for Demand-Driven Structural Testing," *Proc. 14th Int'l Conf. Compiler Construction (CC '05)*, pp. 242-245, 2005.

[26] C. Pacheco and M. Ernst, "Eclat: Automatic Generation and Classification of Test Inputs," *Proc. 19th European Conf. Object-Oriented Programming (ECOOP '05)*, pp. 504-527, 2005.

[27] C. Boyapati, S. Khurshid, and D. Marinov, "Korat: Automated Testing Based on Java Predicates," *Proc. ACM/SIGSOFT Int'l Symp. Software Testing and Analysis (ISSTA '02)*, pp. 123-133, 2002.

[28] C. Csallner and Y. Smaragdakis, "JCrasher: An Automatic Robustness Tester for Java," *Software—Practice and Experience*, vol. 34, no. 11, pp. 1025-1050, 2004.

[29] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Eng.: An Int'l J.*, vol. 10, no. 4, pp. 405-435, 2005.

[30] D. Wheeler, SLOCCount: A Set of Tools for Counting Physical Source Lines of Code, http://www.dwheeler.com/sloccount/, 2006.

[31] K. Apt and E. Olderog, *Verification of Sequential and Concurrent Programs*, second ed. Springer-Verlag, 1997.

[32] D. Engler, D. Chen, and A. Chou, "Bugs as Inconsistent Behavior: A General Approach to Inferring Errors in Systems Code," *Proc. Symp. Operating Systems Principles*, pp. 57-72, 2001.

[33] H. Agrawal, J. Horgan, S. London, and W. Wong, "Fault Localization Using Execution Slices and Dataflow Tests," *Proc. Sixth Int'l Symp. Software Reliability Eng.*, pp. 143-151, 1995.

[34] F. Tip, "A Survey of Program Slicing Techniques," *J. Programming Languages*, vol. 3, pp. 121-189, 1995.

[35] J. Lyle and M. Weiser, "Automatic Program Bug Location by Program Slicing," *Proc. Second Int'l Conf. Computers and Applications*, pp. 877-882, 1987.

[36] Y. Ayalew and R. Mittermeir, "Spreadsheet Debugging," *Proc. European Spreadsheet Risks Interest Group Ann. Conf.*, 2003.

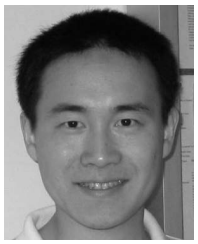
[37] J. Ruthruff, M. Burnett, and G. Rothermel, "An Empirical Study of Fault Localization for End-User Programmers," *Proc. 27th Int'l Conf. Software Eng. (ICSE '05)*, pp. 352-361, 2005.

[38] A. Ko and B. Myers, "Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior," *Proc. SIGCHI Conf. Human Factors in Computing Systems (CHI '04)*, pp. 151-158, 2004.

[39] W. Dickinson, D. Leon, and A. Podgurski, "Finding Failures by Cluster Analysis of Execution Profiles," *Proc. 23rd Int'l Conf. Software Eng. (ICSE '01)*, pp. 339-348, 2001.

[40] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated Support for Classifying Software Failure Reports," *Proc. 25th Int'l Conf. Software Eng. (ICSE '03)*, pp. 465-475, 2003.

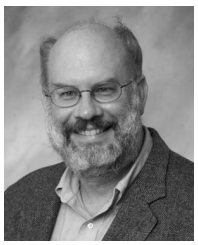
Chao Liu received the BS degree in computer science from Peking University, China, in 2003, and the MS degree in computer science from the University of Illinois at Urbana-Champaign in 2005. He is currently a PhD student in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research focus is on developing statistical data mining algorithms to improve software reliability, with an emphasis on statistical debugging and automated program failure diagnosis. Since 2003, he has more than 10 publications in refereed conferences and journals, such as the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, the International World Wide Web Conference, the European Software Engineering Conference, the ACM SIGSOFT Symposium on the Foundations of Software Engineering, and the *IEEE Transactions on Software Engineering*. He is a member of the IEEE.



Long Fei received the BS degree in computer Science from Fudan University, China, and the MS degree in electrical and computer engineering from Purdue University. He is currently a PhD student in the School of Electrical and Computer Engineering at Purdue University. His research interests are compilers and using compiler techniques for software debugging. He is a member of the IEEE.

Xifeng Yan received the BE degree from the Computer Engineering Department of Zhejiang University, China, in 1997, the MSc degree in computer science from the University of New York at Stony Brook in 2001, and the PhD degree in computer science from the University of Illinois at Urbana-Champaign in 2006. He is a research staff member at the IBM T.J. Watson Research Center. His area of expertise is data mining, with an emphasis on mining and search of graph and network data. His current research is focused on data mining foundations, pattern post analysis, social, biological and Web data mining, and data mining in software engineering and computer systems. He has published more than 30 papers in reputed journals and conferences, such as the *ACM Transactions on Database Systems*, the ACM SIGMOD Conference on Management of Databases, the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, the Very Large Database Conference, the Conference on Intelligent Systems for Molecular Biology, the International Conference on Data Engineering, and the Foundations of Software Engineering Conference. He is a member of the IEEE.

Jiawei Han is a professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. He has been working on research into data mining, data warehousing, stream data mining, spatiotemporal and multimedia data mining, biological data mining, social network analysis, text and Web mining, and software bug mining, with over 300 conference and journal publications. He has chaired or served on many program committees of international conferences and workshops. He also served or is serving on the editorial boards for *Data Mining and Knowledge Discovery*, the *IEEE Transactions on Knowledge and Data Engineering*, the *Journal of Computer Science and Technology*, and the *Journal of Intelligent Information Systems*. He is currently serving as founding editor-in-chief of the *ACM Transactions on Knowledge Discovery from Data* and on the board of directors for the executive committee of the ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD). Jiawei is an ACM fellow and an IEEE senior member. He has received many awards and recognitions, including the ACM SIGKDD Innovation Award (2004) and the IEEE Computer Society Technical Achievement Award (2005).



Samuel P. Midkiff received the PhD degree in 1992 from the University of Illinois at Urbana-Champaign, where he was a member of the Cedar project. In 1991, he became a research staff member at the IBM T.J. Watson Research Center, where he was a key member of the xlHPF compiler team and the Ninja project. He has been an associate professor of computer and electrical engineering at Purdue University since 2002. His research has focused on parallelism, high performance computing, and, in particular, software support for the development of correct and efficient programs. To this end, his research has covered dependence analysis and automatic synchronization of explicitly parallel programs, compilation under different memory models, automatic parallelization, high performance computing in Java and other high-level languages, and tools to help in the detection and localization of program errors. Professor Midkiff has over 50 refereed publications. He is a member of the IEEE.