SOBER: Statistical Model-based Bug Localization[*]

Chao Liu
Dept. Comp. Sci.
University of Illinois-UC
Urbana, IL 61801
chaoliu@cs.uiuc.edu

Xifeng Yan
Dept. Comp. Sci.
University of Illinois-UC
Urbana, IL 61801
xyan@cs.uiuc.edu

Long Fei
School Elec. Comp. Eng.
Purdue University
West Lafayette, IN 47907
lfei@ecn.purdue.edu

Jiawei Han
Dept. Comp. Sci.
University of Illinois-UC
Urbana, IL 61801
hanj@cs.uiuc.edu

Samuel P. Midkiff
School Elec. Comp. Eng.
Purdue University
West Lafayette, IN 47907
smidkiff@ecn.purdue.edu

ABSTRACT

Automated localization of software bugs is one of the essential issues in debugging aids. Previous studies indicated that the evaluation history of program predicates may disclose important clues about underlying bugs. In this paper, we propose a new statistical model-based approach, called SOBER, which localizes software bugs without any prior knowledge of program semantics. Unlike existing *statistical debugging* approaches that select predicates correlated with program failures, SOBER models evaluation patterns of predicates in both correct and incorrect runs respectively and regards a predicate as bug-relevant if its evaluation pattern in incorrect runs differs significantly from that in correct ones. SOBER features a principled quantification of the pattern difference that measures the bug-relevance of program predicates.

We systematically evaluated our approach under the same setting as previous studies. The result demonstrated the power of our approach in bug localization: SOBER can help programmers locate 68 out of 130 bugs in the Siemens suite when programmers are expected to examine no more than 10% of the code, whereas the best previously reported is 52 out of 130. Moreover, with the assistance of SOBER, we found two bugs in bc 1.06 (an arbitrary precision calculator on UNIX/Linux), one of which has never been reported before.

[*]This work was supported in part by the U.S. National Science Foundation NSF ITR-03-25603, IIS-02-09199, and IIS-03-08215. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of the funding agencies.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – Debugging aids, Diagnostics, Testing tools, Tracing

General Terms

Algorithms, Reliability, Experimentation, Verification

Keywords

statistical debugging, localization metrics

1. INTRODUCTION

Despite great advances in software development and testing, software is still far from bug-free. In order to relieve programmers from laborious debugging work, automated bug localization has been extensively studied. Static analysis can detect program defects through checking either a well-specified program model [4] or real code directly [23, 17]. On the other hand, dynamic analysis contrasts the runtime behavior of correct and incorrect executions for isolating suspicious program segments [10, 20, 24, 5, 14, 15]. Dynamic analysis often assumes no prior knowledge of program semantics aside from a labelling of program executions as either correct or incorrect.

Among dynamic methods, statistical bug localization schemes like program invariants [2] and statistical debugging [14], have achieved initial success: programs are first instrumented to collect statistics characterizing their runtime behaviors, such as evaluations of conditionals and function return values. Behaviors can be recorded in the evaluation history of various program predicates. Take the predicate "`idx < LENGTH`" for example, where 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 and analyzed afterwards.

The method described in this study shares the principles of dynamic methods. However, by exploring detailed information about predicate evaluations, our method can detect more subtle bugs than the state-of-the-art approach described by Liblit et al. in [15]. For easy reference, we name their approach Liblit05. For each predicate P in program $\mathcal{P}$, Liblit05 estimates two conditional probabilities respectively: $Pr_1 = Pr(\mathcal{P}\text{ fails}|P\text{ is ever observed})$ 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 bug. Therefore, Liblit05 essentially regards the predicates whose `true` evaluations correlate with program failures as bug-relevant.

Although Liblit05 has succeeded in isolating some bugs in widely used systems [15], it has a potential problem in its ranking model. Because Liblit05 only considers whether a predicate has *ever* been evaluated as `true` in one run, it loses its discrimination power when a predicate P is observed as `true` at least once in every execution. In this case, Pr_1 is equal to Pr_2, which suggests predicate P is not relevant to the bug. In Section 2, we will show an example where a predicate has a small difference between Pr_1 and Pr_2, but is strongly bug-relevant. We also note that cases like the example in Section 2 are not rare, as verified on the Siemens suite in our experiment study.

The above issue motivates us to develop a new approach that exploits the multiple evaluations of a predicate in each program execution. We start from modeling evaluations of predicate P as independent Bernoulli trials: each evaluation of P gives either `true` or `false`. We then estimate the probability of P being `true`, which we call the *evaluation bias*, by calculating the fraction of "`true`" observations within each program execution. While the evaluation biases of P from multiple executions may fluctuate, they can be thought of as observations from an unknown distribution. Moreover, let X be the random variable carrying the evaluation bias of predicate P, there exist two statistical models, $f(X|Correct)$ and $f(X|Incorrect)$, that govern the evaluation bias observed from correct and incorrect executions. Finally, if $f(X|Incorrect)$ significantly differs from $f(X|Correct)$, it indicates that P's evaluation in incorrect runs is quite abnormal, and thus likely related to a bug. Therefore, instead of selecting predicates correlated with program failures as done by Liblit05, our approach statistically models predicate evaluations for both correct and incorrect runs respectively and treats their model difference as a measure of bug-relevance.

In quantifying the difference between $f(X|Correct)$ and $f(X|Failure)$, we propose a hypothesis testing-like approach. Intuitively, it calculates the likelihood that the evaluation biases observed from incorrect runs were generated *as if* from $f(X|Correct)$. Therefore, a small likelihood suggests that evaluations of this predicate in incorrect runs are significantly different from that in correct runs. Using this quantification, we rank all the instrumented predicates, getting a ranked list of suspicious predicates. Programmers can then either examine the list from the top down, or they can choose to examine only the top-ranked ones for debugging.

In summary, we make the following contributions:

1. We propose a probabilistic treatment of program predicates that models how a predicate is evaluated in multiple evaluations during one execution. As discussed in Section 4, this treatment naturally encompasses the program invariants [8] as a special case in software bug isolation.

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 one predicate evaluates in incorrect executions. The more abnormal the evaluations, the more likely a predicate is relevant to the bug.

3. A systematic evaluation of SOBER on the Siemens suite [11, 22] demonstrates that our method greatly advances the state-of-the-art results [5, 15] in software bugs localization w.r.t. minimal code checking. Moreover, as a case study, SOBER also helped us locate two buffer overrun bugs in bc 1.06, one of which has never been reported.

The rest of the paper is organized as follows. Section 2 first provides a motivating example to illustrate the advantages of the probabilistic treatment of predicates over previous approaches. We then develop the statistical models and ranking algorithm in Sections 3 and 4. Systematic evaluations and comparisons are presented in Section 5, after which we describe our case study on bc 1.06. 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 shows the value of modeling predicate evaluations in a probabilistic way. This example motivates our approach of using the differences in evaluation bias between correct and incorrect executions to localize software bugs.

Program 1 Buggy Code - Version 3 of replace

```
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 }
```

Program 1 is excerpted from the faulty version 3 of the replace program in the Siemens suite [11, 22]. The program replace has 512 lines of code (LOC) and performs regular expression matching and substitution. The subclause in Line 7 was intentionally commented out by Siemens researchers to simulate the bug that usually sneaks in when programmers fail to consider fully the `if` condition. Because this is essentially a logic error, which incurs no segmentation faults, even experienced programmers will have few hints as to where the bug is, and will probably resort to conventional debuggers for step-by-step tracing. Our question is: *Can we guide programmers to the buggy place or its neighborhood by contrasting the runtime behaviors between correct and incorrect executions?*

For clarity in what follows, we denote the program with subclause `(lastm != m)` commented out as the incorrect program $\mathcal{P}$, and naturally, the one without comments is the correct one, denoted as $\widehat{\mathcal{P}}$. We understand that $\widehat{\mathcal{P}}$ is definitely not available in debugging $\mathcal{P}$. So $\widehat{\mathcal{P}}$ here is only for illustration purposes: it helps illustrate how our method is motivated. As one will see in Section 3, our method only collects statistics from $\mathcal{P}$ and performs all the analysis.

We declare two boolean variables `A` and `B` as follows.

```
A = (m >= 0);
B = (lastm != 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 to 11) in $\mathcal{P}$ and $\widehat{\mathcal{P}}$. Figure 1 explicitly lists the actions in $\mathcal{P}$ (left) and $\widehat{\mathcal{P}}$ (right), respectively. Clearly, the left panel shows the actual actions taken in the buggy program $\mathcal{P}$, while the right one lists the expected actions if $\mathcal{P}$ had no bugs.

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

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

Figure 1: Branching Actions in $\mathcal{P}$ and $\widehat{\mathcal{P}}$

Differences between the above two tables shown in Figure 1 reveal that in the buggy 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`, control flow should skip the block, but it actually enters the block in $\mathcal{P}$ because the `if` condition A in $\mathcal{P}$ is satisfied. This incorrect control flow usually leads to incorrect outputs. Therefore, in the buggy program $\mathcal{P}$, one run is incorrect if and only if there exist `true` evaluations of $A \wedge \neg B$ in Line 7; otherwise, the execution is correct although the program contains a bug. This explains why not all runs fail in the buggy program.

While the predicate $P : (A \wedge \neg B) = \texttt{true}$ exactly characterizes the scenario under which incorrect executions take place, there is little chance for a bug locator to find P as bug-relevant. The reason is that when we are debugging the incorrect program $\mathcal{P}$, we do not have the correct program $\widehat{\mathcal{P}}$ available. This means that we do not know what B is, let alone its combination with A. Because evaluations of A are observable in $\mathcal{P}$, we are therefore interested in whether the evaluations of A give away the bug. If the evaluations of A in incorrect executions differ significantly from that in correct ones, it may be bug-relevant, and point exactly to the bug location.

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

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

Figure 2: A Correct and Incorrect Run in $\mathcal{P}$

We therefore contrast how an incorrect execution behaves differently from a correct one in the buggy program $\mathcal{P}$. Figure 2 shows the number of `true` evaluations for the four combinations of A and B in one correct (left) and incorrect (right) run. The major difference is that in the correct run, $A \wedge \neg B$ never evaluates as `true` ($n_{A\bar{B}} = 0$) while $n'_{A\bar{B}}$ must be nonzero for one execution to be incorrect. Because $A \wedge \neg B$ evaluates to `true` only in incorrect runs, A has an extra chance to be `true` in incorrect runs. We therefore expect that the probability for A to be `true` is different between correct and incorrect executions.

We tested Program 1 with 5,542 test cases that are available in the Siemens suite. The average probability of A being evaluated as `true` within one incorrect run is 0.9024 and it is 0.2261 in correct ones. This divergence suggests that the bug location (*i.e.*, Line 7) does exhibit detectable abnormal behaviors in incorrect executions. Our method, as developed in Section 3, nicely captures this divergence and ranks $A = \texttt{true}$ as the top bug-relevant predicate. This predicate readily leads the programmer to the bug location. Meanwhile, we note that since neither $A = \texttt{true}$ nor $A = \texttt{false}$ is an invariant in correct or incorrect executions, invariant methods cannot pick up A as a suspicious predicate. Liblit05 cannot detect A as a suspicious predicate either because it does not model the `true` evaluation probability *within* one run (see Section 5.3 for details).

The above example illustrates a simple but representative case where a probabilistic treatment of predicates captures detailed information about predicate evaluations. This information, as we will see soon, can be exploited for effective bug localization. In the next section, we describe our statistical model and ranking algorithm that select bug-relevant predicates.

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. We execute $\mathcal{P}$ on each test case t_i, and obtain 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). We thus partition the test suite T into two disjoint subsets T_p and T_f, corresponding to the passing and failing cases respectively,

$$T_p = \{t_i | o'_i = \mathcal{P}(d_i) \text{ matches } o_i\},$$

$$T_f = \{t_i | o'_i = \mathcal{P}(d_i) \text{ does not match } o_i\}.$$

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

Given a buggy program $\mathcal{P}$ together with a test suite $T = T_p \cup T_f$, our task is to *localize the suspicious bug 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`", "`x != 0`", "`!empty(list)`", etc. Predicate P takes the value `true` or `false` for each evaluation. Considering that a predicate may be evaluated multiple times within one run, we develop the concept of evaluation bias, which measures the probability of a predicate 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* evaluation bias *of predicate P.*

Intuitively, $\pi(P)$ reflects the probability that P takes the value `true` in each evaluation. If P is ever evaluated (*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, 0 if it never holds, and is in between for all other mixtures. 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 assume it is unbiased and set $\pi(P)$ to 0.5. However, if a predicate is never evaluated in any failing runs, it has nothing to do with program failures and is hence eliminated from the following predicate rankings.

3.3 Methodology Overview

In this section, we first lay out the main idea of our method, leaving detailed development to Section 3.4. Following the conventions from statistics, we use uppercase letters for random variables and lowercase letters for their realizations. Moreover, $f(X|\theta)$ is the general notation of probability density function (pdf) given a model θ.

Let the entire test case space be $\mathcal{T}$, which conceptually contains all the possible input and the expected output pairs. According to the correctness of $\mathcal{P}$ on the test cases from $\mathcal{T}$, $\mathcal{T}$ 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 random samples from $\mathcal{T}$, $\mathcal{T}_p$, and $\mathcal{T}_f$ respectively. Given a random test case t from $\mathcal{T}$, let X be the random variable for the evaluation bias of predicate P from the execution of t. We use $f(X|\theta_p)$ and $f(X|\theta_f)$ to denote the probability density function of the evaluation bias of P on $\mathcal{T}_p$ and $\mathcal{T}_f$ respectively. Therefore, the observed evaluation bias of running a test case $t \in \mathcal{T}_p$ is a random sample from $f(X|\theta_p)$; similarly, the observed evaluation bias of running a test case $t \in \mathcal{T}_f$ is a random sample from $f(X|\theta_f)$.

Definition 2 (Bug Relevance). *A predicate P is relevant to a software bug if its underlying density function $f(X|\theta_f)$ differs from $f(X|\theta_p)$, where X is the random variable for the evaluation bias of P.*

The above definition relates $f(X|\theta)$, the distribution of the evaluation bias of predicate P, with the underlying software bug(s). Specifically, a predicate is relevant to a bug if its evaluation distribution for failing runs (*i.e.*, $f(X|\theta_f)$) differs from that for passing runs (*i.e.*, $f(X|\theta_p)$). Moreover, the larger the difference, the more relevant P is to the bug.

Let $\mathbf{L}(P)$ be an arbitrary similarity function,

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

Because we are only interested in the relative ranking of predicates, the ranking score $s(P)$ can be defined as $g(\mathbf{L}(P))$, where $g(x)$ is any monotonically decreasing function. Choosing $g(x) = -log(x)$, the bug relevance score $s(P)$ is thus defined as

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

Using the bug relevance score in Eq. (2), we can rank all of the instrumented predicates. Therefore, the ranking problem boils down to finding a way to quantify the similarity function. This includes two problems: (1) What is a suitable similarity function $\mathbf{L}(P)$, and (2) how is $\mathbf{L}(P)$ computed when the closed form of $f(X|\theta_p)$ and $f(X|\theta_f)$ is unknown? In the following subsections, we examine the two problems in detail.

3.4 Predicate Ranking

In order to quantify the difference between $f(X|\theta_p)$ and $f(X|\theta_f)$, without knowledge about their closed forms, we can only characterize them through general statistics. For instance, for $f(X|\theta_p)$, its mean and variance, $\mu_p = E(X|\theta_p)$ and $\sigma_p^2 = Var(X|\theta_p)$ can be estimated through realized samples,

$$\mu_p = \frac{\sum_{i=1}^{n} x_i}{n}$$

and

$$\sigma_p^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu_p)^2,$$

where x_i is the observed evaluation bias of P in executing $t_i \in T_p$. Similarly, we can get the sample mean and variance of $f(X|\theta_f)$, μ_f and σ_f^2, by running $t \in T_f$.

Generally speaking, it may make sense to combine differences in the mean and variance for the bug relevance score. For example, $s(P)$ can be defined as

$$s(P) = \alpha \cdot |\mu_p - \mu_f| + \beta \cdot |\sigma_p^2 - \sigma_f^2| \qquad (\alpha, \beta \geq 0).$$

However, *ad hoc* solutions like the above usually have the problem of properly setting the parameters, *e.g.*, α and β in the above.

Therefore, in the following, we develop a *principled* approach to quantifying $s(P)$, which is essentially a measure of the difference between $f(X|\theta_p)$ and $f(X|\theta_f)$. Our approach shares a similar rationale as hypothesis testing [13]. Specifically, in order to quantify the difference between $f(X|\theta_p)$ and $f(X|\theta_f)$, we first propose the *null hypothesis* $\mathcal{H}_0$: $\theta_p = \theta_f$. Given a random sample $\mathbf{X} = (X_1, X_2, \cdots, X_m)$ from $f(X|\theta_f)$, we derive, under $\mathcal{H}_0$, a statistic Y, which is expected to conform to a known distribution. If the observation of Y has only small probability to happen, the null hypothesis is likely invalid, which immediately suggests that $f(X|\theta_p)$ and $f(X|\theta_f)$ are not identical.

Formally, we present the null hypothesis $\mathcal{H}_0$:

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

Let $\mathbf{X} = (X_1, X_2, \cdots, X_m)$ be an *independent and identically distributed (i.i.d.)* random sample from the density function $f(X|\theta_f)$. Under the null hypothesis, we have $E(X_i) = \mu_p$ and $Var(X_i) = \sigma_p^2$. Since $X_i \in [0, 1]$, both $E(X_i)$ and $Var(X_i)$ are finite. According to the Central Limit Theorem [3], the following statistic

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

conforms to $N(\mu_p, \frac{\sigma_p^2}{m})$ as $m \to +\infty$.

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 $L(\theta_p|Y)$ of θ_p given an observation 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. This, in consequence, indicates a larger difference between $f(X|\theta_p)$ and $f(X|\theta_f)$. Therefore, we can reasonably set the similarity function in Eq. (1) as the likelihood function, *i.e.*,

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

According to the property of normal distribution, the normalized statistic

$$Z = \frac{Y - \mu_p}{\sigma_p/\sqrt{m}} \quad (7)$$

conforms to the standard normal distribution $N(0, 1)$. Therefore

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

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

Combining Eq. (2), (6), (5), and (8), we finally derive the ranking score for predicate P as

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

We note that the above score exhibits the asymptotic behavior of a random sample as the sample size $m \to +\infty$. Practices in statistical inferences suggest that the asymptotic behavior is still approximately valid even when the sample size is nowhere near infinity [3]. In our debugging scenario, the realized sample $\mathbf{x} = (x_1, x_2, ..., x_m)$ corresponds to the observed evaluation biases from the m failing runs available. It is definitely true that we cannot have infinite failing runs in practice. However, as shown in experiments, Eq. (9) still works well in ranking abnormal predicates even when we have only a small number of failing runs.

4. GENERALIZING INVARIANTS

Without loss of generality, predicate P is a program invariant [8] if and only if it always takes the `true` evaluation. According to DEFINITION 1, the evaluation bias of invariants is always 1. Let $f(X|\theta_p)$ be the population of evaluation bias for an invariant P, then $\mu_p = 1$ and $\sigma_p = 0$. The following theorem proves that the score function in Eq. (9) can identify invariant violations and conformations.

THEOREM 1. *Let P be an invariant summarized from correct executions. $s(P) = +\infty$ if there is a violation in incorrect executions and $s(P) = -\infty$ if the invariant P is conformed in incorrect executions.*

PROOF. Given a random sample $\mathbf{x} = (x_1, x_2, \cdots, x_m)$, which corresponds to the observed evaluation bias from m failing runs. Once there exists at least one run where invariant P is violated, $\sum_{i=1}^{m} x_i \neq m$. It then follows from Eq. (7) that

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

then

$$\begin{aligned}\lim_{\sigma_p \to 0} \frac{\sigma_p}{\sqrt{m}\varphi(z)} &= \sqrt{\frac{2\pi}{m}} \lim_{\sigma_p \to 0} \frac{\sigma_p}{e^{-\frac{1}{2}(\frac{c}{\sigma_p})^2}} = \sqrt{\frac{2\pi}{m}} \lim_{t \to \infty} \frac{e^{\frac{c^2t^2}{2}}}{t} \\ &= c^2\sqrt{\frac{2\pi}{m}} \lim_{t \to \infty} te^{\frac{c^2t^2}{2}} = +\infty.\end{aligned}$$

Thus Eq. (9) gives $s(P) = +\infty$. Intuitively, our method treats violated invariants as the most abnormal predicates and ranks them at the top.

On the other hand, if failing runs do not violate the invariant, we have

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

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 the least abnormal and are ranked at the bottom by our method. □

Theorem 1 indicates that if a bug can be caught by invariant violations as implemented in the DIDUCE [9] project, our method can also detect it because the bug relevant score for a violated invariant is $+\infty$. Meanwhile, as to conformed invariants, our method simply ignores them due to their $-\infty$ scores. While previous research [19] has suggested that invariant violations themselves can only locate a limited number of bugs in the Siemens suite, our ranking algorithm, being a superset of invariant-based methods, actually achieves the best bug localization results.

5. EXPERIMENTAL RESULTS

In this section, we evaluate the effectiveness of our statistical model-based bug localization algorithm, SOBER, and compare it with two state-of-the-art bug localization algorithms: Cause Transition (CT) proposed by Cleve and Zeller [5], and the statistical approach (Liblit05) by Liblit et al. [15]. We subject these three algorithms to a standard testbed, the Siemens suite [11, 22], and evaluate the localization quality with an objective measure as described in Section 5.1.

The Siemens suite was originally prepared by Siemens Corp. Research in study of test adequacy criteria [11]. A variant is available at http://www.cc.gatech.edu/aristotle/Tools/subjects. The Siemens suite contains seven programs: print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info. For each program, Siemens researchers manually injected multiple bugs, getting multiple faulty versions, and each faulty version has one bug. The Siemens suite contains 130 faulty versions in total, which simulate a wide spectrum of realistic bugs. Due to its high quality, many researchers investigating bug localization have reported their results on it [10, 19, 20, 5]. Readers interested in details about the Siemens suite are referred to [11, 22].

In the following, Section 5.1 first introduces some necessary background on performance metrics we use. We briefly review CT and Liblit05 in Sections 5.2 and 5.3 respectively and compare SOBER with them from Section 5.5.

5.1 Performance Metrics

Performance metrics are always a central issue in accurate and objective comparisons. In measuring bug localization quality, we here adopt a framework that is based on the program static dependencies. This measure was originally proposed by Renieris et al. [20] and was later adopted by Cleve et al. in reporting the performance of CT [5]. We briefly summarize this measure as follows.

1. Given a (buggy) program $\mathcal{P}$, its program dependence graph is written as $\mathcal{G}$, where each statement is a node and there is an edge between two nodes if two statements have data and/or control dependencies.

2. The buggy statements are marked as *defect* nodes. The set of defect nodes is written as V_{defect}.

3. Given a bug localization report R, which is a set of suspicious statements, their corresponding nodes are called *blamed* nodes. The set of blamed nodes is written as V_{blamed}.

4. A programmer can start from V_{blamed} and perform the breadth-first search until he reaches one of the defect nodes. 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 bug,

$$T = \frac{|V_{examined}|}{|V|} * 100\%, \tag{10}$$

where $|V|$ is the size of the program dependence graph $\mathcal{G}$. In [20, 5], the authors used $1 - T$ as a measure.

T-score roughly measures the real cost in locating a bug. The less code to be examined, the higher the quality of a bug report R. A good algorithm should generate a high quality bug report requiring minimal code checking. For algorithms that generate a ranked list of all predicates, users can select the top-k most suspicious as a report. The best strategy for the optimal value of k is to choose the one under which a bug can be located with minimum code checking, *i.e.*,

$$k_{opt} = \operatorname*{argmin}_{k} \ \mathrm{E}[T_k], \tag{11}$$

where $\mathrm{E}[T_k]$ is the average value of T-Score for a set of bugs under study given a fixed value of k.

5.2 Cause Transition Algorithm: CT

The Cause Transition algorithm [5], denoted as CT, is an enhanced variant of Delta Debugging proposed by Zeller [24]. The original Delta Debugging compares the memory graph [25] of a failing execution e_f against the memory graph of a passing execution e_p. Through manipulating the memory contents of these two executions, Delta Debugging systematically narrows down the differences between e_f and e_p to a small set of suspicious variables. CT enhances Delta Debugging through exploiting *cause transitions*: "moments where new relevant variables begin being failure causes" [5]. This actually implements the concept of "search in time" in addition to the original "search in space".

5.3 Statistical Debugging: Liblit05

Liblit et al. proposed a statistical approach to ranking predicates according to their correlation with program crashes [15]. The top ranked predicates are considered as hints for debugging. We name this approach Liblit05. Liblit05 contrasts the probability that one execution crashes when a predicate is *ever* observed `true` and when a predicate is observed (either `true` or `false`). Specifically, the authors define

$$\begin{aligned} Context(P) &= Pr(Crash|P \textit{ observed}) & (12)\\ Failure(P) &= Pr(Crash|P \textit{ observed } \texttt{true}), & (13) \end{aligned}$$

and treat the probability difference

$$Increase(P) = Failure(P) - Context(P), \tag{14}$$

as a measure of the extent, to which predicate P is related to the underlying bug(s). Finally, the ranking score in [15] also considers the number of failing runs where P is ever observed as `true`.

A detailed examination reveals fundamental differences between Liblit05 and SOBER. First, from the methodological point of view, Liblit05 evaluates how much more likely one execution crashes if the predicate P is observed as `true` than that when P is only observed. This means that Liblit05 essentially values predicates whose `true` evaluations correlate with program crashes. SOBER, on the other hand, models the evaluation distribution of the predicate P in passing (*i.e.*, $f(X|\theta_p)$) and failing (*i.e.*, $f(X|\theta_f)$) runs respectively and regards predicates with large differences between $f(X|\theta_f)$ and $f(X|\theta_p)$ as bug-relevant. Therefore, SOBER and Liblit05, actually follow two fundamentally different approaches although both of them adopt statistical analysis in ranking predicates. Second, SOBER explores the multiple evaluations of predicates *within one execution* while Liblit05 disregards them. For instance, if a predicate P evaluates as `true` at least once in each execution but has different probabilities to be `true` in correct and incorrect runs, Liblit05 simply ignores P while SOBER readily captures the evaluation abnormality in incorrect execution.

Let us re-examine Program 1 in Section 2. The buggy statement (Line 7) is executed almost in every execution. Within one run, it takes multiple evaluations as `true` and `false`. In this case, Liblit05 has little discrimination power. In detail, for predicate P : "$(m >= 0)$ = `true`", $Increase(P) = 0.0104$ and the $Increase$ value for predicate P' : "$(m >= 0)$ = `false`" is -0.0245. According to [15], neither P nor P' is ranked top since they are either negative or too small. Thus, Liblit05 fails to identify the defect point in Program 1. As we experimented, SOBER successfully ranks P as the most suspicious predicate. Intuitively, SOBER works because the evaluation bias in failing runs (0.9024) significantly diverges from that in passing runs (0.2261).

5.4 Implementations

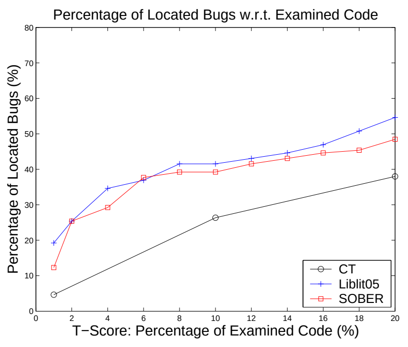
All of the experiments in this section were carried out on a 3.2GHz Intel Pentium-4 PC with 1GB physical memory, running Fedora Core 2. We implemented both SOBER and Liblit05 within Matlab. Because this work focuses on the *comparison* of bug localization quality, no sampling is taken in collecting predicate evaluations at runtime. Intuitively, analyses from full execution traces should reflect the best performance for both methods.

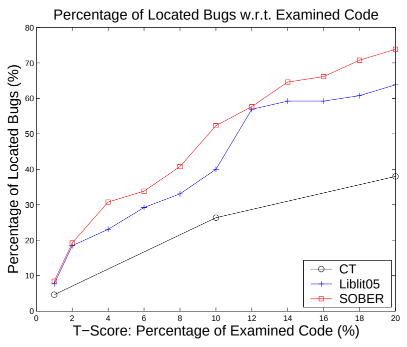
We instrumented the source code to monitor predicates of three categories: **branches**, **returns** and **scalar-pairs**, which are described in detail in [14, 15]. With preliminary experiments, we observed that dropping **scalar-pairs** can generally reduce by half the overhead without significantly degrading the localization quality. Therefore, only **branches** and **returns** were finally instrumented for experiments.

In the evaluations, we generated program dependence graphs using CodeSurfer 1.9 with patch 3 and the gcc compiler 3.3.3. Because evaluation results may vary with different build switches in CodeSurfer, we finally chose to take the factory default (by enabling the `factory-default` switch) so that results can be replicated in the future.

5.5 Performance Comparison

In this section, we compare our method SOBER with CT


(a) Top-1


(b) Top-5

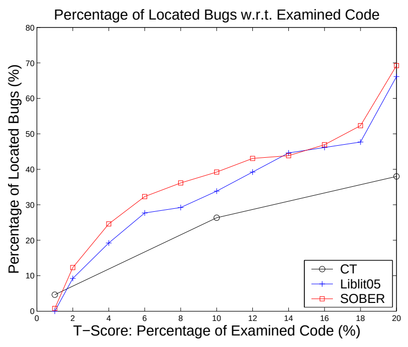

(c) Top-10

Figure 5: Performance Comparison w.r.t. Various top-k Values

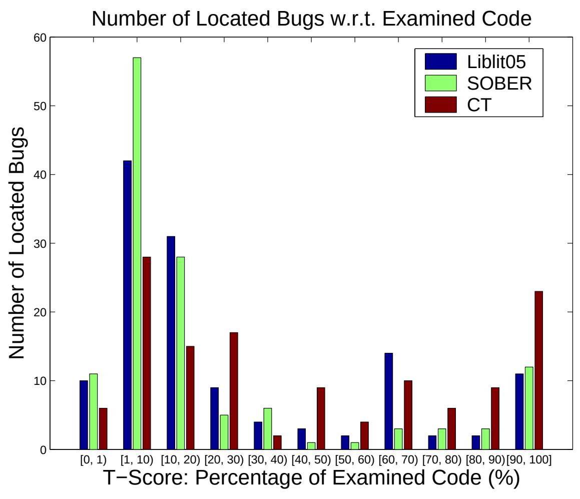


Figure 3: Located Bugs w.r.t. Code Examination

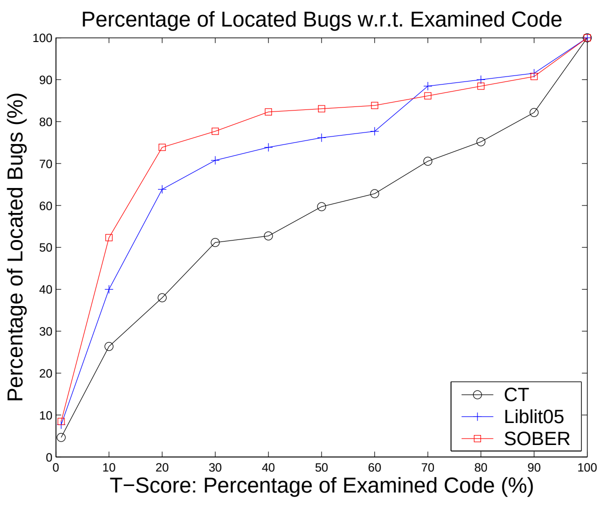


Figure 4: Cumulative Comparison

and Liblit05. We subject both Liblit05 and SOBER to the 130 bugs in the Siemens suite and measure their localization quality using the T-Score (Eq. (11)). The result of CT is directly cited from [5].

Figure 3 depicts the number of bugs that can be located when a certain percentage of code is examined by a programmer. The x-axis is labelled with T-Score, which is the percentage of code to be examined. For Liblit05 and SOBER, we choose the top-5 most suspicious statements to form the set of blamed nodes. Because reports that require programmers to examine more than 20% of the code are likely useless, we take the T-Score range $[0, 20]$ as the meaningful range. Figure 3 shows that 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) bugs can be identified when no more than $\alpha\%$ code is examined. We therefore plot the cumulative comparison in Figure 4. It clearly suggests that both SOBER and Liblit05 are much better than CT and that SOBER consistently outperforms Liblit05. Although Liblit05 catches up in the T-Score range [60, 100], we regard this advantage as irrelevant because it does not make much sense for a bug locator to require more than 60% code examination.

In detail, Figure 4 tells us that for the 130 bugs in the Siemens suite, when a programmer would like to examine at most 1% of the code, CT catches 4.65% of the bugs while Liblit05 and SOBER capture 7.69% and 8.46% respectively. Moreover, when 10% code examination is acceptable, CT and Liblit05 identify 34 (26.36%) and 52 (40.00%) out of the 130 bugs. Our method SOBER, being the best of the three, locates 68 (52.31%) out of 130 bugs, catching 16 more bugs than the state-of-the-art approach Liblit05. When the user is patient enough to examine 20% of the code, 73.85% of the bugs (*i.e.*, 96 out of 130) can be located by SOBER.

5.6 The Optimal Setting of K

As we discussed in Section 5.1, the selection of top-k suspicious nodes will affect the performance of both Liblit05 and SOBER. We plot the performance curve for each algorithm with $k = 1$, $k = 5$, and $k = 10$ in Figure 5. We confine the comparison within the [0, 20] meaningful T-score range with finer x ticks. Since we do not have detailed results from [5], CT is still only depicted at the 1, 10, and 20 ticks. Figure 5 shows that Liblit05 is the best when $k = 1$, and SOBER apparently outperforms the other two when $k = 5$. Finally, when $k = 10$, SOBER is slightly better than Liblit05. Since users are always interested in locating bugs with min-

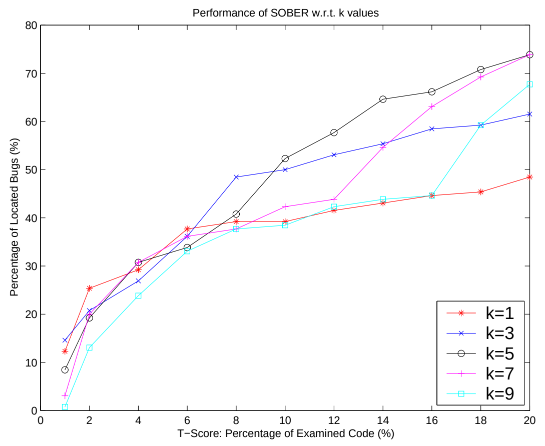


Figure 6: Performance of SOBER w.r.t. top-k Values

imal code examinations, it is desirable to select the optimal k that maximizes the performance. We found that both Liblit05 and SOBER achieve their best performance when k is equal to 5 as shown in Figure 5. In addition, Figure 6 plots the performance of SOBER with various k-values. It clearly indicates that SOBER finds the most bugs when k is set to 5. Therefore, the setting of $k = 5$ in Figures 3 and 4 is justifiable. Figure 6 also suggests that too few suspicious points (*e.g.*, $k = 1$) may not convey enough information for bug localization while too many (*e.g.*, $k = 9$) are in themselves a burden for programmers to examine, and thus both lead to poor localization qualities.

5.7 Time Complexity

Our method SOBER is computationally efficient in that it scores each predicate once according to Eq. (9) and sorts all of the predicates at the end. Given a program under study, suppose we collect the runtime statistics of k predicates from n correct and m incorrect executions, the time complexity of scoring each predicate is $O(n+m)$. Therefore, the entire time complexity for SOBER is $O((n+m) \cdot k + k \cdot log(k))$. Similarly, because Liblit05 also needs $O(n+m)$ to score each predicate, it has the same time complexity $O((n+m) \cdot k + k \cdot log(k))$. We experimented with the 31 faulty versions of replace, which has 512 LOC, the average time for Liblit05 and SOBER to analyze each version is 11.7775 and 11.3844 seconds respectively. This speed is much faster than CT [5].

5.8 Advantages of CT

Although comparisons above indicate that both Liblit05 and SOBER outperform CT in quality and efficiency, they should not be carelessly interpreted as showing either Liblit05 or SOBER is always a better substitute for CT. CT has its own unique advantages. In the first place, CT only needs one pair of passing and failing runs for analysis, although some work is needed in selecting a proper pair. Liblit05 and SOBER, on the other hand, rely on multiple runs to obtain reliable statistics. Moreover, the localization report produced by CT generally contains more information than those generated by Liblit05 and SOBER. In particular, CT not only provides a set of suspicious points but tries to relate them in the form of a cause-effect chain as well. This makes a bug report from CT more accessible to the programmer. Therefore, SOBER and Liblit05 are essentially complementary to, rather than a substitute for CT. For instance, when a programmer encounters one failing case during development, CT is a good choice to use; when multiple failing runs are available (*e.g.*, during regression testing or beta-testing), Liblit05 and SOBER can be expected to work better.

6. CASE STUDY: BC 1.06

Systematic evaluations in Section 5 demonstrate the effectiveness of SOBER in comparison with CT and Liblit05. Although the Siemens suite covers a wide spectrum of bugs through its 130 faulty versions, its subject programs are mainly small-scale: ranging from 141 to 512 LOC [11]. In this section, we report on a case study of SOBER on a real-world program bc 1.06: SOBER precisely identifies two buffer overflow bugs, one of which has never been reported before. More extensive case studies are in progress.

bc is a calculator program that accepts scripts written in the bc language, a numeric processing language supporting arbitrary precision. bc 1.06 in this study is shipped with most of recent UNIX/Linux distributions. It has 14,288 LOC. A single buffer overflow error was once reported in [14, 15].

This experiment was conducted on a P4 3.06 GHz machine running Linux RedHat 9 with gcc 3.3.3. Inputs to bc 1.06 are 4,000 randomly generated valid bc programs with various sizes and complexities. We generate each input program in two steps: First, a random syntax tree is generated which complies with the bc language specification; second, a program is derived from the syntax tree.

Program 2 bc 1.06, storage.c

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

Program 3 bc 1.06, 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;
      ...
    }
```

With the aid of SOBER, we quickly identify two bugs in bc 1.06, including a previously unreported one. Among the

4,000 input cases we test on bc 1.06, there are 3,479 correct cases and 521 incorrect ones. After running through these cases, the analysis from SOBER reports "indx < old_count" as the highest ranked predicate. This predicate points to old_count in line 137 in storage.c as shown in Program 2. A quick scan of the code shows that old_count copies the value from v_count. By putting a watch on v_count using gdb, we quickly find that v_count is overwritten when a buffer named genstr overflows (in bc.y, line 306). genstr is an 80-byte-long buffer used to hold byte code characters. An input containing complex and relatively large functions can easily overflow this buffer[1]. To the best of our knowledge, this bug has never been reported before. We manually examined the statistics of this top-ranked predicate and found its evaluation biases in correct and incorrect executions are 0.0274 and 0.9423 respectively, which explains why SOBER worked so well. Liblit05 also ranks this predicate at the top, and locates the bug.

After we fix the first bug, a second run of SOBER (3303 correct and 697 incorrect cases) generates a bug report with the top predicate "a_count < v_count" pointing to line 176 in storage.c as shown in Program 3. This is most likely a copy-paste error where a_count should have been used in the position of v_count. This bug was reported in previous studies [14, 15]. SOBER values the predicate "a_count < v_count" because its evaluation biases differ significantly between passing and failing executions: 0.0224 in passing and 0.5157 in failing runs. Although the ultimate explanation of predicate ranking is Eq. (9), this divergence in the evaluation bias does shed light on why SOBER is effective.

As a final note, predicates identified by SOBER for these two bugs are far from the actual crashing points. This suggests that SOBER picks up predicates that characterize the scenario under which bugs are triggered, rather than the crashing scenes, which are simply available with conventional debuggers, like gdb.

7. DISCUSSION

7.1 Related Work

In this section, we briefly review previous works related to bug detection in general. Static analysis techniques have been used to verify program correctness against a well-specified program model [1, 4] and to check real codes directly for Java [23] and C/C++ programs [17]. Engler et al. also show that checking programmers' beliefs against themselves can be an effective approach to bug detection [7]. Complementary to static analysis, dynamic analysis focuses more on the runtime behavior checking and often assumes few specifications. SOBER belongs to the category of dynamic analysis.

Within dynamic analysis, the DAIKON project automatically discovers likely invariants from executions of instrumented programs [8]. It first poses a wide spectrum of predicates and gradually rejects or relaxes those violated. Predicates that are valid in all runs are taken as invariants. The DIDUCE project [9] 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 project relates further violations as indications of potential bugs. Recent studies also show that program invariants can be used to find latent errors [2] and to warn about unsafe upgrades [16], to name just a few. However, because invariants hold through all test runs, they may not be effective in locating subtle bugs as [19] suggests. In comparison, our probabilistic treatment of predicates naturally relaxes this requirement and is shown to achieve the best performance on the Siemens suite.

Program spectra, proposed by Reps et al. in [21], also work well for bug localization. Earlier work in [10] explores the relation between spectra differences and faults in regression testing. Similar information is later visualized in [12], making it more accessible to the programmers. Recent researches by Renieris et al. show that spectrum differences are more effective for bug localization when it is applied along with nearest neighbor queries [20]. Taking program spectra as a summary of program runtime behaviors, we find that the above program spectra-based methods can be thought of as model-based approaches: The spectra are taken from passing and failing run(s) respectively and their differences suggest the bug location. In comparison, our method models general predicates and quantifies the model difference using statistical principles.

The power of statistical analysis is demonstrated in program analysis and bug detection. Dickinson et al. find program failures through clustering program execution profiles [6]. Their subsequent work [18] first performs feature selection using logistic regression and in consequence clusters failure reports within the space of selected features. Cluster results are shown to be useful in prioritizing software bugs. Early work of Liblit et al. on statistical debugging [14] 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.

7.2 Threats to Validity

Care should be taken in interpreting the experiment results and conclusions thus drawn. The first threat lies in the benchmark selection. Since the Siemens suite mainly contains small-scale subject programs, *absolute* performance measures (*e.g.*, 52.31% bugs are located with at most 10% code examination) do not readily generalize to arbitrarily large programs. On the other hand, because the Siemens suite covers a wide variety of bugs, the *relative* performance comparison is *statistically significant*, and hence credible.

Results in this study are also subjected to the threat of performance metrics. Although the evaluation framework based on the program dependence graphs involves few subjective judgements, it is by no means a comprehensively fair metric. For instance, this measure does not take into account how easily a programmer can make sense of the bug localization report. Recent work [5] also identifies some other limitations of this measurement. However, to the best of our knowledge, this is by far the most objective metric for bug localization besides large-scale user studies.

8. CONCLUSIONS

In this paper, we proposed a statistical approach to lo-

[1]Some random inputs with fewer than 50 LOC can overflow the buffer. The bc specification allows a parser depth of up to 150 levels. Our random inputs have much smaller depth.

calize software bugs without prior knowledge of program semantics. This approach tackles the limitation of previous methods in modeling the divergence of predicate evaluations between correct and incorrect executions. Systematic evaluations through the Siemens suite, together with a case study in `bc 1.06`, clearly demonstrated the advantages of our method in bug localization. A number of interesting topics merit further study. It is not yet clear how the localization quality varies along with insufficient test cases. Moreover, it is also instructive to know how robust SOBER is to evaluation samplings.

Acknowledgements

We would like to thank the insightful discussions with Darko Marinov. Gregg Rothermel made the Siemens suite available. Holger Cleve and Manos Reneris generously provided us with their evaluation frameworks. GrammaTech Inc. offered us a free copy of CodeSurfer . We greatly appreciate the anonymous reviewers who not only offered us detailed review comments but also insightful suggestions on future work.

9. REFERENCES

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

[2] Y. Brun and M. Ernst. Finding latent code errors via machine learning over program executions. In *Proc. of the 26th Int. Conf. on Software Engineering (ICSE'04)*, pages 480–490, 2004.

[3] G. Casella and R. Berger. *Statistical Inference*. Duxbury Press, second edition, 2001.

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

[5] H. Cleve and A. Zeller. Locating causes of program failures. In *Proc. of the 27th Int. Conf. on Software Engineering (ICSE'05)*, 2005.

[6] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proc. of the 23rd Int. Conf. on Software Engineering (ICSE'01)*, pages 339–348, 2001.

[7] D. Engler, D. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symp. on Operating Systems Principles*, pages 57–72, 2001.

[8] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Software Engineering*, 27(2):1–25, 2001.

[9] S. Hangal and M. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. of the 24th Int. Conf. on Software Engineering (ICSE'02)*, 2002.

[10] 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 & Reliability*, 10(3):171–194, 2000.

[11] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of the 16th Int. Conf. on Software Engineering (ICSE'94)*, pages 191–200, 1994.

[12] J. Jones, M. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proc. of the 24th Int. Conf. on Software Engineering (ICSE'02)*, pages 467–477, 2002.

[13] E. Lehmann. *Testing Statistical Hypotheses*. Springer, second edition, 1997.

[14] B. Liblit, A. Aiken, A. Zheng, and M. Jordan. Bug isolation via remote program sampling. In *Proc. of ACM SIGPLAN 2003 Int. Conf. on Programming Language Design and Implementation (PLDI'03)*, pages 141–154, 2003.

[15] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *Proc. of ACM SIGPLAN 2005 Int. Conf. on Programming Language Design and Implementation (PLDI'05)*, 2005.

[16] S. McCamant and M. Ernst. Predicting problems caused by component upgrades. In *Proc. of the 9th European Software Engineering Conf. held jointly with 11th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering (ESEC/FSE'03)*, pages 287–296, 2003.

[17] M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. Cmc: A pragmatic approach to model checking real code. In *Proc. of the 5th Symp. on Operating System Design and Implementation (OSDI'02)*, 2002.

[18] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proc. of the 25th Int. Conf. on Software Engineering (ICSE'03)*, pages 465–475, 2003.

[19] B. Pytlik, M. Renieris, S. Krishnamurthi, and S. Reiss. Automated fault localization using potential invariants. In *Proc. of the 5th Int. Workshop on Automated and Algorithmic DebuggingSymp (AADEBUG'03)*, pages 287–296, 2003.

[20] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries. In *Proc. of the 18th IEEE Int. Conf. on Automated Software Engineering (ASE'03)*, 2003.

[21] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. *SIGSOFT Softw. Eng. Notes*, 22(6):432–449, 1997.

[22] G. Rothermel and M. Harrold. Empirical studies of a safe regression test selection technique. *IEEE Trans. on Software Engineering*, 24(6):401–419, 1998.

[23] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. of the 15th IEEE Int. Conf. on Automated Software Engineering (ASE'00)*, 2000.

[24] A. Zeller. Isolating cause-effect chains from computer programs. In *Proc. of ACM 10th Int. Symp. on the Foundations of Software Engineering (FSE'02)*, 2002.

[25] T. Zimmermann and A. Zeller. Visualizing memory graphs. In *Revised Lectures on Software Visualization, International Seminar*, pages 191–204, 2002.