Reducing Confounding Bias in Predicate-Level Statistical Debugging Metrics

Ross Gore and Paul F. Reynolds, Jr.
Dept. of Computer Science
University of Virginia
Charlottesville, VA, USA
{*rjg7v,reynolds*}*@virginia.edu*

***Abstract*—Statistical debuggers use data collected during test case execution to automatically identify the location of faults within software. Recent work has applied causal inference to eliminate or reduce control and data flow dependence confounding bias in statement-level statistical debuggers. The result is improved effectiveness. This is encouraging but motivates two novel questions: (1) how can causal inference be applied in predicate-level statistical debuggers and (2) what other biases can be eliminated or reduced. Here we address both questions by providing a model that eliminates or reduces control flow dependence and failure flow confounding bias within predicate-level statistical debuggers. We present empirical results demonstrating that our model significantly improves the effectiveness of a variety of predicate-level statistical debuggers, including those that eliminate or reduce only a single source of confounding bias.**

***Keywords*-automated debugging; fault localization**

I. Introduction

Recently, there has been considerable research on using statistical approaches for fault localization [1]–[14]. These approaches, referred to as *statistical debuggers*, require test inputs, corresponding execution profiles, and a labeling of the test executions as either succeeding or failing. The execution profiles reflect coverage of individual statements, the truth-values of branches or other inserted predicates (conditional propositions).

In the canonical predicate-level statistical debugger Cooperative Bug Isolation (CBI), three predicates are inserted and tested for each assignment statement to, or return of, a variable k: $(k > 0)$, $(k = 0)$ and $(k < 0)$ [1], [2]. Within the predicate-level statistical debugger Exploratory Software Predictor (ESP), these three predicates are complemented with elastic predicates. Elastic predicates use profiling to compute the mean (μ_k), and standard deviation (σ_k), of the values assigned to, or returned from a variable k. Using these profiled statistics, the CBI predicates are complemented with elastic predicates such as: $(k = \mu_k)$, $(k > \mu_k + \sigma_k)$ and $(k < \mu_k + \sigma_k)$ [14].

Once formed, the predicates in ESP and CBI are scored for suspiciousness. Then developers examine the predicates in decreasing order of suspiciousness until the fault is discovered. One metric used to score predicate suspiciousness is the probability of a program Q failing given that a predicate p is true. This probability, Pr(Q fails $\mid p = true$), indicates if the condition specified by predicate p was true during an execution of Q.

Given the execution of a test suite, Pr(Q fails $\mid p = true$) is typically estimated by the *specificity* metric, $\frac{f_p}{(f_p+s_p)}$, where f_p is the number of tests for which p is true and the program fails and where s_p is the number of tests for which p is true and the program succeeds. Approaches that estimate the probability Pr(Q fails $\mid p = true$) use statistical techniques on observational data to determine the effect of individual predicates on program failures. However, the suspiciousness metrics used in these approaches can be susceptible to biases.

Recently Baah et. al. showed that *control flow dependence confounding bias* exists within statement-level suspiciousness metrics [10], [11]. *Confounding bias* occurs when an apparent causal effect of an event on an outcome may actually be due to an unknown confounding variable, which causes both the event and the outcome [15], [16]. Baah et al. showed that coverage of the immediately preceding dependent statement in the control flow, the *forward control flow predecessor*, can cause dependent statements to contribute to a program's failure and that existing metrics do not account for this *control flow confounding bias*.

By accounting for *control flow confounding bias* bias at the statement-level, Baah et al. improved the effectiveness for a variety of established statement-level statistical debugging suspiciousness metrics. More recently, Baah et al. improved their statement-level approach by also controlling for confounding bias caused by data flow dependences between statements [11].

Here we look to adapt Baah et al.'s work for predicate-level statistical debuggers. The adaptation has challenges requiring innovation. One of the contributions of our work is a robust method to efficiently track the *forward control flow predecessor predicate* and employ it in a causal inference model that eliminates *control flow confounding bias* in different predicate-level statistical debugging metrics.

Within predicate-level statistical debugging approaches the issue of bias has been previously explored. Liblit et al. showed that the estimate $\frac{f_p}{(f_p+s_p)}$ of Pr(Q fails $\mid p = true$) is biased [1], [2]. The estimate is biased because once the fault in a program has been triggered, the probability of the program failing is 1.0, thus the observations collected from

978-1-4673-1067-3/12/$31.00 © 2012 IEEE

ICSE 2012, Zurich, Switzerland

subsequent predicates are more susceptible to failure. This reflects *failure flow confounding bias*.

Liblit et al. proposed the suspiciousness metric, *Importance*, as a correction for *failure flow confounding bias*. *Importance* measures the suspiciousness of a predicate p, not by the chance that it implies failure, but by how much difference it makes that the predicate p is observed to be true versus simply reaching the line where the predicate p is evaluated [1]. Within *Importance*, Pr(Q fails $\mid p = true$) is estimated by the difference: $\frac{f_p}{(f_p+s_p)}$ - $\frac{f_{p\,obs}}{(f_{p\,obs}+s_{p\,obs})}$.

Here, $f_{p\,obs}$ and $s_{p\,obs}$ are the number of respective failing and succeeding test runs for which p is reached and evaluated (true or false). This correction attempts to factor out predicates that are more susceptible to failure because of the program flow once the fault is triggered. While this heuristic can be effective it is not a proven solution.

The second contribution of our work is a causal inference model which accounts for *failure flow confounding bias* at the predicate-level. The combination of our two contributions yields a predicate-level statistical debugging metric that is significantly more effective than existing metrics because the *control flow* and *failure flow* confounding biases are reduced and/or eliminated.

These contributions are needed. While control flow and data flow dependence biases are evident at the statement-level and the predicate-level, failure flow is only distinguishable at the predicate-level [1], [2]. We present an empirical evaluation showing that our model significantly improves the effectiveness of a variety of predicate-level statistical debugging metrics in two different predicate-level statistical debuggers.

II. Motivating Example

An example helps elucidate how confounding bias occurs in predicate-level suspiciousness metrics. Consider the procedure `distance()` in Figure 1, which has a fault in Statement 5. The procedure should print the one dimensional Euclidean distance between two points x and y. However, for some test cases `distance()` returns a negative number. The observational data collected for the truth of the predicates for statements 5, 6, 7 and 8 is shown in Table I.

The first two columns in Table I identify the statement number and condition tested in each of the predicates. The third through the seventh columns identify the inputs for five test cases. The '1/0' entires within these columns indicate if the corresponding predicate was true in each test case. An entry of '1' means that the predicate was true and an entry of '0' means that the predicate was not true. The bottom row of Table I shows the Boolean outcome of executing each test case. A failing test case execution is denoted by 'F' and a successful (or passing) test case execution is denoted by 'S'.

The two rightmost columns (SR and CR) in Table I indicate the rank of each predicate based on two different

```
1  int
2  distance(int x, int y)
3  {
4       int diff = x - y;
5       if (!( diff > 1)){ /* off by one */
6           int dist = 0;
7           dist = y - x;
8           print(dist);
9       }
10      int dist = 0;
11      dist = x - y;
12      return dist;
13 }
```

Figure 1. Faulty function that calculates euclidean distance between two one-dimensional points.

Table I
Test Cases and Predicate Data for Figure 1.

LoC	Predicate	2,2	5,4	5,1	-4,-2	1,0	SR	CR
5	`diff = 0`	1	0	0	0	0	12	12
5	`diff > 0`	0	1	0	0	1	***3***	***1***
5	`diff < 0`	0	0	1	1	0	12	12
6	`dist = 0`	1	1	1	1	1	4	12
6	`dist > 0`	0	0	0	0	0	12	12
6	`dist < 0`	0	0	0	0	0	12	12
7	`dist = 0`	1	0	0	0	0	12	12
7	`dist > 0`	0	0	1	1	0	12	12
7	`dist < 0`	0	1	0	0	1	***3***	2
8	`dist = 0`	1	0	0	0	0	12	12
8	`dist > 0`	0	0	1	1	0	12	12
8	`dist < 0`	0	1	0	0	1	***3***	2
S/F	———	S	F	S	S	F	–	–

suspiciousness metrics. The first metric is the *specificity* metric described in the Introduction. The second metric, found in the rightmost column of Table I, is derived from a causal model which *controls for* the effects of other predicates on the suspiciousness of the predicate being estimated and on the occurrence of program failure.

Table I shows that the *specificity* metric identifies three predicates which rank as the most suspicious predicates in the procedure `distance()` shown in Figure 1. These predicates are: $(\texttt{diff} > 0)_5$, $(\texttt{dist} < 0)_7$ and $(\texttt{dist} < 0)_8$. A predicate x specifying a condition corresponding to program Statement y is abbreviated as x_y. The first predicate, $(\texttt{diff} > 0)_5$, reflects the location of the fault in the procedure `distance()`. However, the two other predicates have the same suspiciousness rank as $(\texttt{diff} > 0)_5$ and correspond to innocent statements. These two predicates have the same rank as the fault localizing predicate, despite their ties to innocent statements, because their truth is dependent on the condition `diff > 0` in Statement 5. This dependence causes the predicates $(\texttt{dist} < 0)_7$ and $(\texttt{dist} < 0)_8$ to be true in every failing execution. Since the *specificity* metric *does not control for* dependence among

predicates, confounding bias exists and the two dependent predicates $(\texttt{dist} < 0)_7$ and $(\texttt{dist} < 0)_8$ receive the same rank as the fault localizing predicate $(\texttt{diff} > 0)_5$. This dependency related confounding bias is referred to as *control flow confounding bias*.

However, another type of confounding bias exists in the *specificity* metric used to rank the predicates in the procedure `distance()`. Additional confounding bias exists because the fault in Statement 5 is triggered before the predicates corresponding to statements 6, 7 and 8 are evaluated. This bias is evident in the *specificity* suspiciousness rank (4) calculated for the predicate $(\texttt{dist} = 0)_6$, which corresponds to the innocent declaration of the variable `dist`.

While the condition `dist = 0` in Statement 6 is uncorrelated with failure, it is always true immediately following the execution of Statement 5 and thus is true in every failing test case. As a result the *specificity* metric finds the predicate $(\texttt{dist} = 0)_6$ to be suspicious. However, it is important to note that the chance of a test case failing given that $(\texttt{dist} = 0)_6$ is true, is the same as the chance of a test case failing given that Statement 6 is executed. The inability to control for the difference between the chance of a test case failing given that a predicate is *true* and the chance of a test case failing given that the predicate is *evaluated* (true or false) results in a second type of confounding bias: *failure flow confounding bias*.

The factors that create *control flow* and *failure flow* confounding bias can be controlled for in a causal model. The causal model produces a suspiciousness metric with reduced bias that more effectively ranks predicates for fault localization. In this example, the causal model predicate rankings clearly identify one suspicious predicate that localizes the fault in the procedure `distance()` in Figure 1. This is shown in the right-hand column of Table I. In Section III the factors that create *control flow* and *failure flow* confounding bias are identified and causal models which produce suspiciousness metrics with reduced bias are presented.

III. Controlling for Confounding Bias

Within the motivating example in Section II, two different types of confounding bias are evident: *control flow* and *failure flow*. While controlling for both of these confounding biases through a causal model to estimate suspiciousness for predicate-level statistical debuggers is our novel work, each of the biases has been previously explored, separately, in other research.

At the statement-level, Baah et al. have shown that in a subject program *control flow confounding bias* is manifested on a given statement *stmt* through the execution of the statement immediately preceding *stmt* in the control flow graph. This statement is the *forward control flow predecessor* of *stmt*. By employing a causal model to control for the *forward control flow predecessor*, Baah et al. improved the effectiveness of suspiciousness metrics for statement-level statistical debuggers [10], [11]. In Section III-C we adapt Baah et al.'s statement-level work on reducing *control flow confounding bias* to the predicate-level.

While *control flow confounding bias* had not been perviously explored at the predicate-level, *failure flow confounding bias* has. Liblit et al. have shown that *failure flow confounding bias* exists in the *specificity* metric and proposed the *Importance* measure to address it. Recall, the *Importance* measure estimates Pr(Q fails | p=true) with the difference $f_p/(f_p + s_p) - f_{p\,obs}/(f_{p\,obs} + s_{p\,obs})$. The terms $f_{p\,obs}$ and $s_{p\,obs}$ are the number of respective failing and succeeding test cases for which p is evaluated. Liblit et al.'s estimate of Pr(Q fails | p=true) measures not the chance that a predicate p implies failure, but how much difference it makes that the predicate p is observed to be true versus simply reaching the line where the predicate p is evaluated [1], [2]. This correction is a heuristic to factor out the program flow once the fault is triggered within a subject program. While Liblit et al.'s heuristic can be effective it does not employ a methodology, such as causal inference, that is a proven solution for addressing confounding bias.

In Section III-D, our causal model which reduces *control flow* and *failure flow* confounding bias is presented. The evaluation in Section IV shows that employing this causal model to control for both confounding biases results in significantly more effective suspiciousness metrics for predicate-level statistical debuggers than any existing suspiciousness metrics, including Liblit et al.'s *Importance* estimate.

However, before any of our models for reducing confounding bias are presented, Section III-A and Section III-B review background information related to suspiciousness metrics, observational studies, confounding bias and causal inference which is required to understand our approach.

A. Existing Suspiciousness Metrics

The extent to which a predicate p reflects subject program failure is measured through a *suspiciousness metric*. The following terms are used in existing predicate-level metrics to estimate suspiciousness: s is the total number of tests that succeed (pass), f is the total number of tests that fail, s_p is the number of tests that succeed where p is true and f_p is the number of tests that fail where p is true. Different suspiciousness metrics use these terms differently. Here, several established suspiciousness metrics are reviewed.

1) Tarantula: The *Tarantula* suspiciousness metric, shown in Eq. 1, is closely related to the *specificity* metric,

$$Tarantula = \frac{\frac{f_p}{f}}{\frac{f_p}{f} + \frac{s_p}{s}}. \tag{1}$$

Tarantula differs from *specificity* because it includes the number of failed test cases, f, in the numerator and denominator and it includes the number of successful test cases, s,

in the denominator. However, when $s = f$ as shown in Eq. 2, *Tarantula* reduces to the *specificity* metric [6],

$$Tarantula = \frac{\frac{f_p}{f}}{\frac{f_p}{f} + \frac{s_p}{s}} = \frac{\frac{f_p}{f}}{\frac{f_p}{f} + \frac{s_p}{f}} = \frac{f_p}{f_p + s_p}. \quad (2)$$

2) F_1: F_1 balances an estimate of Pr(Q fails | p=true) with an estimate of the probability that predicate p is true given that the subject program Q failed, Pr(p = true | Q fails). The *sensitivity* metric, $\frac{f_p}{f}$, is used in the F_1 measure to estimate Pr(p = true | Q fails) [7].

The inclusion of *sensitivity* within a suspiciousness metric addresses issues that occur when a predicate p is true in very few failing test cases. Without *sensitivity*, if there are many successful (passing) test cases where p is true, the overall sample of test cases where p is true will be unbalanced and the *specificity* will be low even if there are very few failed test cases. Conversely, if there are few successful (passing) test cases where p is true, the overall sample of test cases where p is true will be small and the *specificity* could be high even if there are many failed test cases where predicate p is not true [7],

$$F_1 = \frac{2}{\frac{1}{\frac{f_p}{f}} + \frac{1}{\frac{f_p}{f_p + s_p}}}. \quad (3)$$

3) Importance: Liblit's *Importance* measure is closely related to the F_1 metric [1], [2]. However, the *specificity* metric used in the F_1 metric is replaced with the difference: $f_p/(f_p + s_p) - f_{p\,obs}/(f_{p\,obs} + s_{p\,obs})$. This difference is referred to as the *Increase*. The first term in *Increase* is identical to the *specificity* metric. However, the second term in *Increase* is meant to ensure that predicate p is scored, not by the chance that p implies failure, but by how much difference it makes that p is true versus simply reaching the statement where p is evaluated (true or false). Formally, the *Importance* measure, shown in Eq. 4, is

$$Importance = \frac{2}{\frac{1}{\frac{f_p}{f}} + \frac{1}{Increase}}. \quad (4)$$

4) Ochiai: Other statistics besides the harmonic mean can be employed to balance the metrics *specificity* and *sensitivity*. The *Ochiai* metric, shown in Eq. 5, balances *specificity* and *sensitivity* with the geometric mean [7],

$$Ochiai = \sqrt{\frac{f_p}{f} \times \frac{f_p}{f_p + s_p}}. \quad (5)$$

B. Observational Studies and Confounding Bias

Reducing or eliminating confounding bias within observational studies is a well studied research topic. In this section, we view predicate-level statistical debugging as an observational study and a novel approach to reduce confounding bias is summarized.

Casting predicate-level statistical debugging as an observational study yields two groups of test case executions. These two execution groups are: those where predicate p is true (the $treatment$ group) and those where predicate p is not true (the $control$ group) [15]. For a predicate p, the membership of a test case in either the $treatment$ or the $control$ group is denoted by the treatment variable T_p. For those test cases where predicate p is true, T_p = 1. For those test cases where predicate p is not true, T_p = 0. Independent of the presence of a given predicate, test case executions can also be classified with an outcome variable Y. A successful test case execution is denoted by Y=0 and a failing test case execution is denoted by Y=1.

In the context of an observational study, estimating the average treatment effect of a predicate on a test case corresponds to estimating the probability of program Q failing given that a specific predicate p is true, Pr(Q fails | p=true). The average treatment effect of a predicate, τ_p, is estimated by a regression model. The most basic estimate for the average treatment effect of a predicate is computed using the regression model in Eq. 6. The model is linear and is solved with least-squares regression, α_p is an intercept and σ_p is a random error term that is uncorrelated with T_p [17]. The estimate of Pr(Q fails | p=true) yields the same ranks as the *specificity* metric.

1) For each predicate p in a faulty subject program Q fit a separate model M_p according to the following:
   a) The *outcome* variable Y is 1 for a test case if it fails and is 0 otherwise.
   b) The *treatment* indicator T_p is 1 for a test case if p is true and is 0 otherwise.
2) Rank the predicates in descending order of, $\tau_{ls,p}$. $\tau_{ls,p}$ is the least-squares estimate, for each predicate, of the coefficient of the treatment variable T_p in the model M_p.

$$Y = \alpha_p + \tau_p T_p + \epsilon_p \quad (6)$$

The model shown in Eq. 6 shows the symmetry between the *specificity* metric and the estimate of the average treatment effect of a predicate on subject program test case outcomes. However, the model in Eq. 6 ignores any dependencies between the treatment variable T_p and the outcome variable Y. This relationship reflects *treatment selection*. Ignoring *treatment selection* is equivalent to assuming the treatment variable T_p and the outcome variable Y are independent. The metrics in Section III-A make this assumption. As a result, they suffer from confounding bias.

It is often possible to characterize the process of *treatment selection* in terms of one or more important variables. If a set of covariates accounts well for which units in an observational study receive treatment and which do not, then it is possible to reduce or eliminate confounding bias when estimating the average treatment effect. In the context of statistical debugging, if a set X of covariates accounts well for those test cases where a given predicate p is true

(T=1) and those test cases where p is not true (T=0), then it is possible to reduce or eliminate confounding bias in the suspiciousness estimate for a predicate. This is accomplished by controlling for (or conditioning on) X in a model that estimates the average treatment effect of a predicate [15]. In the next sections models which employ this approach to control for different covariates are presented.

C. Control Flow Dependency Confounding Bias

Our predicate-level adaptation of Baah et al.'s statement-level work for controlling for *control flow confounding bias* begins with defining and identifying the *forward control flow predecessor statement* for a given predicate. This requires a review of control flow graphs and statement dependency.

A program's control flow graph is a directed graph whose nodes correspond to program statements and whose edges represent control dependences between statements [18]. Node Y is *control dependent* on node X if X has two outgoing edges and the traversal of one edge always leads to the execution of Y while the traversal of the other edge does not necessarily execute Y. Node X *dominates* node Y in a control flow graph if every path from the entry node to Y contains X. Node Y is *forward control dependent* on node X if Y is control dependent on X and Y does not dominate X [18]. Forward control dependences are control dependences that can be realized during execution without necessarily executing the dependent node more than once. Node X is a *forward control flow predecessor* of Node Y if Y is *forward control dependent* on X and X immediately precedes Y in the control flow graph. The statement corresponding to node X is the *forward control flow predecessor statement* of the statement corresponding to node Y that is defined by Baah et al. [1], [2].

We identify the *forward control flow predecessor predicate* using an approach that is similar to Baah et al.'s algortihm to find the *forward control flow predecessor statement*. For a given predicate p, corresponding to statement $stmt$, both approaches extract the control-dependence graph of a subject program and identify the control flow statement in the graph immediately preceding *stmt*. This is the *forward control flow predecessor* for *stmt*.

However, identifying the *forward control flow predecessor* for p requires an additional step. Recall, a statement reflects a line of source code while predicates represent conditions that are true about variables within the line of source code. Thus, given a *forward control flow statement*, the *forward control flow predecessor predicate* is located by instrumenting the *forward control flow predecessor statement* with two branch predicates. The first of the two predicates asserts that the branch is false. The second of the two predicates asserts that the branch is true. When the branch is reached, exactly one of the two predicates will be true. The predicate that is true is the *forward control flow predecessor predicate*.

This approach is applicable to all two-way branches in subject programs. Two-way branching statements include: `if` statements, branches governing `for` and `while` loops, branches implied by the logical '`&&`' and '`||`' operators and implicit branches. However, multiple-way `switch` statements are rarely two-way branches. As a result, each branch of a `switch` statement is instrumented with a predicate which asserts that the branch is true. When the `switch` statement is reached exactly one of the branches and one of the predicates will be true. The predicate that is true is the *forward control flow predecessor predicate*. Branch predicate instrumentation is not a contribution of our work, however, innovatively employing branch predicates to capture *forward control flow predecessor predicates* is our contribution.

The ability to capture the *forward control flow predecessor predicate* for a given predicate enables our model which controls for *control flow dependency* confounding bias to be defined. Given an instrumented subject program, and the feedback reports from executing a set of test cases with a predicate-level statistical debugger such as CBI or ESP, the following model reduces or eliminates *control flow confounding bias*:

1) For each predicate p in a faulty model Q fit a separate linear model M_p according to the following:
   a) The *outcome* variable Y is 1 for a test case if it fails and is 0 otherwise.
   b) The *treatment* variable T_p is 1 for a test if p is true and is 0 otherwise.
   c) If p has a *forward control flow predecessor predicate*, $cfp(p)$, then M_p has a single binary covariate C_p, which is 1 for a test case if $cfp(p)$ is true and is 0 otherwise. If p does not have a *forward control flow predecessor predicate* then M_p has no covariates.
2) Rank the predicates in descending order of, $\tau^c_{ls,p}$, the least-squares estimate, for each predicate, of the coefficient of T_p in M_p.

The resulting linear model for a predicate p is:

$$Y = \alpha^c_p + \tau^c_p\, T_p + \beta^c_p\, C_p + \epsilon^c_p \tag{7}$$

The coefficient $\tau^c_{ls,p}$ is the average treatment effect of predicate p on subject test case outcomes. This estimate is a reduced bias version of the *specificity* metric.

The role of covariate C_p is to control for confounding bias of the suspiciousness estimate for predicate p, due to the truth of other subject program predicates. Intuitively, conditioning on C_p reduces confounding bias because $cfp(p)$ is the most immediate cause of p being evaluated or p not being evaluated in a particular test case.

Pearls *Back-Door Criterion* for causal graphs provides a formal, causal inference justification for the model in Eq. 7 [16]. For any subject program where failure is determined

with a single output statement and control dependences carry all the causal influences of failure, any back door paths from a predicate to the output statement must begin with the *forward control flow predecessor predicate*, $cfp(p)$. Thus, *cfp(p)* is a suitable covariate of T_p because it satisfies Pearl's Back-Door Criterion. It blocks all *back-door paths* in the dynamic control flow graph from the predicate corresponding to the treatment variable T_p to output statement corresponding to the outcome variable Y. As a result the *control flow dependency confounding bias* in, $\tau^{c}_{ls,p}$, the model's estimate of the average treatment effect, is reduced.

D. Failure Flow Confounding Bias

Recall, Liblit et al.'s *Importance* measure employs the difference $f_p/(f_p+s_p) - f_{p\,obs}/(f_{p\,obs}+s_{p\,obs})$ as a heuristic to reduce *failure flow confounding bias*. The term measures, not the chance that a predicate p implies failure, but how much difference it makes that the predicate p is observed to be true versus simply reaching the line where the predicate p is evaluated [1]. While this can be an effective means of reducing *failure flow confounding bias*, it is a heuristic, not a proven solution. Formally reducing or eliminating *failure flow confounding bias* requires a causal model where the set of covariates within the model satisfy the causal inference *Back-Door Criterion*. The difference, $f_p/(f_p + s_p) - f_{p\,obs}/(f_{p\,obs} + s_{p\,obs})$, does not meet these requirements. In this subsection, our model, which controls for predicate evaluation (true or false), satisfies Pearl's *Back-Door Criterion* and reduces *failure flow confounding bias* is presented.

Given the outcomes and predicate coverage vectors from executing a set of test cases with a predicate-level statistical debugger such as CBI or ESP the following approach reduces or eliminates both *control flow* and *failure flow* confounding biases:

1) For each predicate p in a faulty model Q fit a separate linear model M_p according to the following:
   a) The *outcome* variable Y is 1 for a test case if it fails and is 0 otherwise.
   b) The *treatment* variable T_p is 1 for a test if p is true and is 0 otherwise.
   c) If p has a *forward control flow predecessor predicate*, $cfp(p)$, then M_p has a single binary covariate C_p, which is 1 for a test case if $cfp(p)$ is true and is 0 otherwise. If p does not have a *forward control flow predecessor predicate* then M_p does not have this covariate.
   d) The covariate D_p is 1 for a test case if p is evaluated (true or false) and is 0 otherwise.
2) Rank predicates in descending order of, $\tau^{c,f}_{ls,p}$, the least-squares estimate, for each predicate, of the coefficient of T_p in M_p.

The resulting linear model for a predicate p is:

$$Y = \alpha_p + \tau^{c,f}_p\, T_p + \beta^{c,f}_p\, C_p + \omega^{c,f}_p\, D_p + \epsilon_p \qquad (8)$$

Here, the coefficient $\tau^{c,f}_{ls,p}$ is the least-squares estimate of the average treatment effect of a predicate p on the subject program test case outcomes. The notable difference between Eq. 8 and the causal model shown in Eq. 7 is the inclusion of the covarite D_p, which reflects whether or not predicate p is evaluated. Intuitively, D_p further reduces confounding bias because it enables the model to determine how much difference it makes that the predicate p is observed to be true versus simply reaching the line where the predicate p is evaluated.

Once again, inclusion of the covariate C_p blocks all back-door paths from the predicate corresponding to the treatment variable T_p to the output statement corresponding to the outcome variable Y. This enables the additional source of bias controlled for by D_p to be reduced without introducing other sources of bias into $\tau^{c,f}_{ls,p}$. This is significant. Without the inclusion of covariate C_p in the model to block all back-door paths, D_p would not be a suitable covariate.

E. Improving Existing Suspiciousness Estimates

The reduced bias metrics derived in sections III-C and III-D can be integrated into the suspiciousness metrics presented in Section III-A. Recall, that *Tarantula* and the reduced bias metrics $\tau^{c}_{ls,p}$ and $\tau^{c,f}_{ls,p}$ are similar to *specificity*. Thus, integrating either $\tau^{c}_{ls,p}$ and $\tau^{c,f}_{ls,p}$ into the *Tarantula* metric is a straightforward substitution.

It is slightly more difficult to integrate $\tau^{c}_{ls,p}$ and $\tau^{c,f}_{ls,p}$ into the *Ochiai*, F_1 and *Importance* metrics because these measures also employ *sensitivity*. As a result, both $\tau^{c}_{ls,p}$ and $\tau^{c,f}_{ls,p}$ must be converted into a probability value before either can replace the *specificity* measure in the *Ochiai*, F_1 or *Importance* metrics. The *inverse logit* function, $\exp(x)/(1+\exp(x))$, is used to convert $\tau^{c}_{ls,p}$ and $\tau^{c,f}_{ls,p}$ into a probability value. The function constrains the value of the reduced bias estimates to the range 0.0 - 1.0 and ensures the combination with the *sensitivity* measure is meaningful.

Once converted $\tau^{c}_{ls,p}$ and $\tau^{c,f}_{ls,p}$ can be substituted into the *Ochiai*, F_1 and *Importance* metrics in place of the biased *specificity* estimate. Performing this substitution in the F_1 metric and the *Importance* measure renders the two metrics indistinguishable because the only term in which they differ is replaced with $\tau^{c}_{ls,p}$ or $\tau^{c,f}_{ls,p}$.

IV. Evaulation

A. Experimental Setup

The utility of a statistical debugging approach is determined through empirical evaluation using established benchmarks. Characteristics of the benchmarks included in our evaluation are listed in Table II. Each was obtained from [19], except bc, which was obtained from [20].

Table II
EVALUATION BENCHMARKS

Name	LoC	Vers.	Tests	Description
tcas	138	41	1608	altitude seperation
totinfo	396	23	1052	information measure
schedule	299	9	2650	priority queue
schedule2	297	9	2710	priority queue
print-tokens	472	5	4130	lexical analyzer
print-tokens2	399	10	4115	lexical analyzer
replace	512	31	5542	pattern recognition
sed	6,092	7	363	stream editing utility
space	14,382	35	157	ADL interpreter
bc	14,288	1	4,000	basic calculator
gzip	7,266	9	217	compression utility

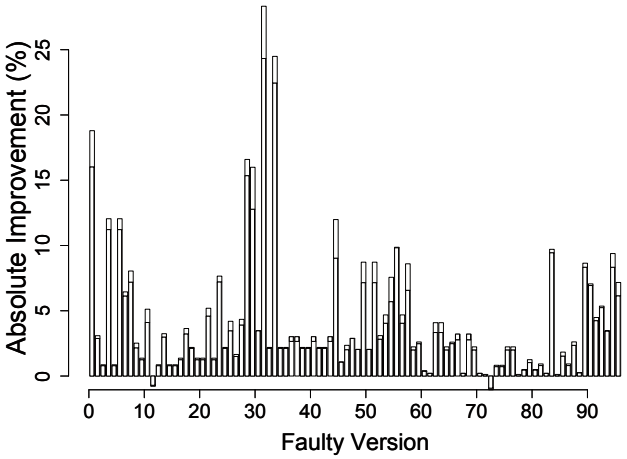
The Siemens Suite consists of seven benchmarks and 132 faulty versions. In our evaluation we omitted four versions: version 32 of replace, version 9 of schedule2, and versions 4 and 6 of print-tokens. We omitted these versions because either there were no syntactic differences between the correct version and the faulty versions of the program or none of the test cases failed when executed on the faulty version of the program.

The space program has 38 faulty versions and several different coverage-based test suites. We used 35 of the 38 faulty versions. For these versions we found a test suite that achieved branch-coverage and resulted in a combination of passing and failing tests cases. We had difficulty finding such a test suite for the remaining three versions. There are seven versions of the sed program with multiple faults per version that can be activated separately. We activated one fault for each of the seven versions. bc is a calculator program with a reported buffer overflow fault [3]. Our bc test suite is comprised of 4,000 valid randomly generated programs with various sizes and complexities. gzip is a well known compression utility with an established test suite [19].

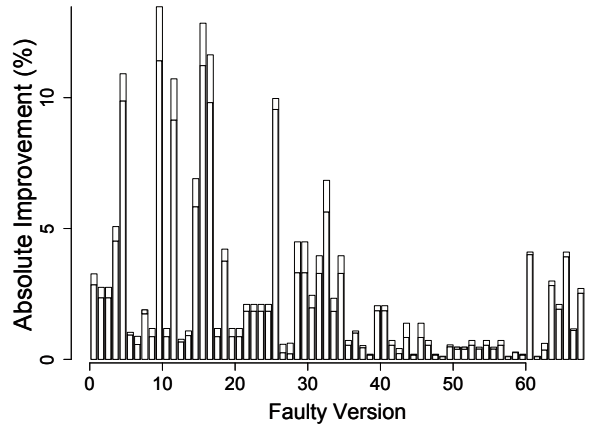
In total, we evaluated 180 faulty versions. For each test case, using the CIL Framework, we computed feedback reports to communicate test case success (or failure), predicate-truth and predicate-evaluation [21]. Also using CIL, we computed the dynamic control flow graph for each function in each version. Our approach uses the feedback reports and control flow graphs as inputs. The regression models and the existing suspiciousness metrics are implemented in the statistical language R [22].

B. Ranking Effectiveness

To measure the effectiveness of the suspiciousness metrics we use an established cost-measuring function (*Cost*) [1]–[8], [10]–[14], [23]. *Cost* measures the percentage of predicates a developer must examine before the faulty statement is found, assuming the predicates are presented in descending order of suspiciousness. To compare two metrics A and B



(a) Standard *Tarantula* as the reference metric.

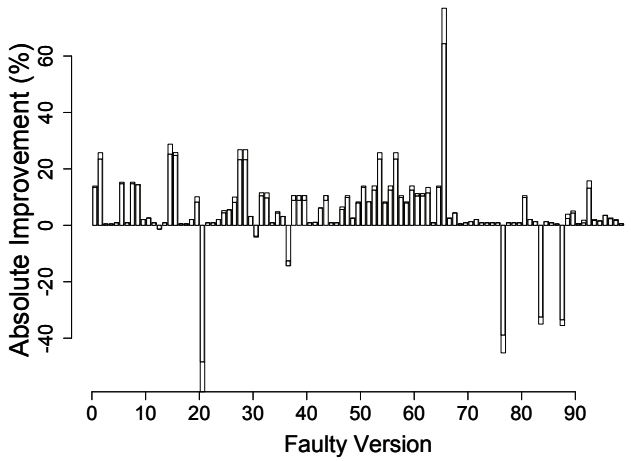


(b) $\tau^{c}_{ls,p}$ as the reference metric.

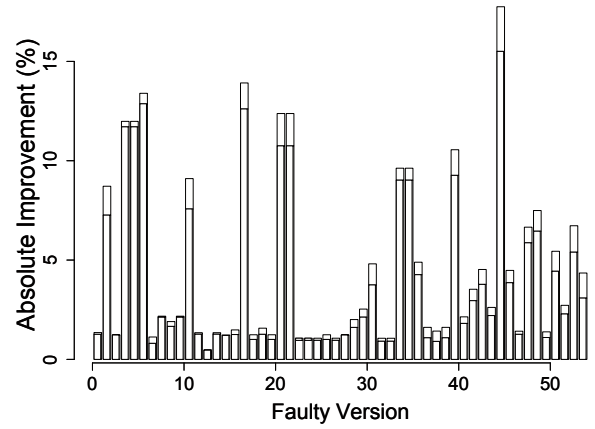
Figure 2. *Tarantula* Evaluation.

for effectiveness, we choose one metric, (B), as the reference metric and subtract the *Cost* value for A from the *Cost* value for B. If A performs better than B, then the *Cost* is positive and if B performs better than A, the *Cost* is negative. For example, for a given program, if the *Cost* of A is 30% and the *Cost* of B is 40%, then the *absolute improvement* of A over B is 10% because developers would examine 10% fewer predicates using A instead of B.

We integrate the reduced bias metrics, $\tau^{c}_{ls,p}$ and $\tau^{c,f}_{ls,p}$, into the *Tarantula*, F_1 and *Ochiai* metrics. We evaluate each reduced bias metric in CBI (red) and ESP (white). For each program version, the absolute improvement of each reduced bias metric within each predicate-level debugger is represented with a bi-colored bar. The height of the colored portion of the bar closest to the x-axis reflects the



(a) Standard F_1 as the reference metric.



(b) F_1 integrated with $\tau^c_{ls,p}$ as the reference metric.

Figure 3. F_1 Evaluation.

improvement for the matching debugger. The total height of both portions reflects the improvement for the debugger matching the colored portion furthest from the x-axis.

1) Predicate-level Tarantula Suspiciousness Metric: Here, we evaluate the effectiveness of the two reduced bias metrics $\tau^c_{ls,p}$ and $\tau^{c,f}_{ls,p}$ compared to the standard *Tarantula* suspiciousness metric in the predicate-level statistical debuggers CBI and ESP. $\tau^c_{ls,p}$ and $\tau^{c,f}_{ls,p}$ are obtained from the models presented in Eq. 7 and Eq. 8, respectively. Fig. 2(a) shows that the reduced *control flow confounding bias* estimate, $\tau^c_{ls,p}$, performs better than the standard *Tarantula* metric on 94 program versions within CBI and ESP but it performs worse on two versions. $\tau^c_{ls,p}$ performs the same as the standard *Tarantula* metric on 84 versions. Fig. 2(b) compares $\tau^c_{ls,p}$ and $\tau^{c,f}_{ls,p}$. It uses $\tau^c_{ls,p}$ as the reference metric and measures the *Cost* of $\tau^c_{ls,p}$ subtracted from the *Cost* of $\tau^{c,f}_{ls,p}$. $\tau^{c,f}_{ls,p}$ performs better than $\tau^c_{ls,p}$ for 68 of the 180 versions and it never performs worse.

2) Predicate-level F_1 Suspiciousness Metric: Here, we evaluate the effectiveness of $\tau^c_{ls,p}$ and $\tau^{c,f}_{ls,p}$ within the F_1 metric in CBI and ESP. Fig. 3(a) shows that the F_1 metric employing $\tau^c_{ls,p}$ performs better than the F_1 metric using the standard specificity measure on 92 versions within CBI and ESP, but performs worse on seven versions. The metrics perform the same on 92 versions. Fig. 3(b) compares F_1 metric employing $\tau^c_{ls,p}$ to the version $\tau^{c,f}_{ls,p}$. Once again, the metric employing $\tau^{c,f}_{ls,p}$ outperforms the metric using $\tau^c_{ls,p}$. The F_1 metric employing $\tau^{c,f}_{ls,p}$ performs better than the F_1 metric employing $\tau^c_{ls,p}$ for 54 of the 180 programs and it never performs worse.

3) Predicate-level Ochiai Suspiciousness Metric: Here, we evaluate the effectiveness of $\tau^c_{ls,p}$ and $\tau^{c,f}_{ls,p}$ within the *Ochiai* metric in CBI and ESP. Fig. 4(a) shows that the *Ochiai* metric employing $\tau^c_{ls,p}$ performs better than the *Ochiai* metric using the standard specificity measure on 59 versions within CBI and ESP, but performs worse on two versions. The metrics perform the same on 119 versions. Fig. 4(b) compares *Ochiai* metric employing $\tau^c_{ls,p}$ to the version using $\tau^{c,f}_{ls,p}$. Again, the metric employing $\tau^{c,f}_{ls,p}$ outperforms the metric using $\tau^c_{ls,p}$. The *Ochiai* metric employing $\tau^{c,f}_{ls,p}$ performs better than the *Ochiai* metric employing $\tau^c_{ls,p}$ for 41 of the 180 programs and it never performs worse.

4) Discussion: Throughout the evaluation, the suspiciousness metric employing $\tau^{c,f}_{ls,p}$ is the most effective metric within both CBI and ESP. These results show that *control flow* and *failure flow* confounding bias exist in established predicate-level suspiciousness metrics and that our metric, $\tau^{c,f}_{ls,p}$, reduces or eliminates these confounding biases.

Space precludes a graphical comparison of $\tau^{c,f}_{ls,p}$ with the *Importance* measure. However, we evaluated *Importance* against the reduced bias versions of the *Tarantula*, F_1 and *Ochiai* metrics employing $\tau^{c,f}_{ls,p}$. Table III shows that in the preponderance of the program versions the metric employing $\tau^{c,f}_{ls,p}$ outperforms the *Importance* metric.

Furthermore, in Section IV-D the F_1 metric employing $\tau^{c,f}_{ls,p}$ is evaluated favorably against the *Importance* metric at different predicate sampling rates. Thus for CBI and ESP and the evaluated programs, the suspiciousness metrics employing $\tau^{c,f}_{ls,p}$ are superior to the *Importance* measure.

Although the metrics $\tau^c_{ls,p}$ and $\tau^{c,f}_{ls,p}$ performed well in our evaluation, each was not as effective as the standard metric for some program versions. The faults in these versions violate the *coverage trigger assumption*, which assumes that the coverage of a statement corresponding to the predicate p will necessarily trigger a failure, if the statement is faulty [10]. However, covering a faulty statement corresponding to predicate p may not be sufficient to trigger a failure because the statement does not cause an invalid internal state or

Table III
METRICS EMPLOYING IMPORTANCE VS. $\tau^{c,f}_{ls,p}$.

Metric	Better than *Importance*	Same as *Importance*	Worse than *Importance*
Tarantula	102	30	48
F_1	110	41	29
Ochiai	114	38	28

Table IV
CBI AND ESP RELATIVE EFFICIENCY.

Metric	Standard (CBI,ESP)	$\tau^{c}_{ls,p}$ (CBI,ESP)	$\tau^{c,f}_{ls,p}$ (CBI,ESP)
Tarantula	1.00,1.00	1.94, 1.24	2.16, 1.36
F_1	1.05,1.02	1.97, 1.25	2.17, 1.37
Ochiai	1.06,1.04	1.99, 1.25	2.17, 1.37

the invalid state does not propagate to the programs output. Often these faults correspond to missing statements where the predicates corresponding to statements adjacent to the missing code qualify as the fault.

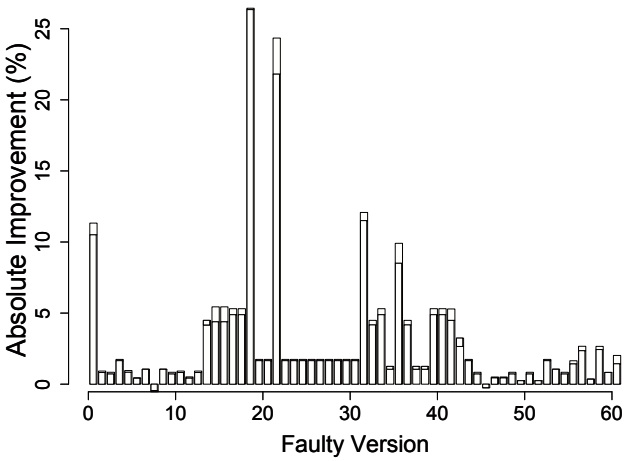
The effectiveness of $\tau^{c}_{ls,p}$ and $\tau^{c,f}_{ls,p}$ relative to CBI and ESP is also important to discuss. For the preponderance of the program versions CBI offers more improvement than ESP. However, for most of the program versions ESP incurs less overall Cost for developers. This paradox can be explained. ESP has been shown to be more effective than CBI when standard biased suspiciousness metrics are employed [14]. While $\tau^{c}_{ls,p}$ and $\tau^{c,f}_{ls,p}$ improve the effectiveness of each predicate-level debugger, ESP appears to improve less by absolute measure because of its superior effectiveness. Similarly, for most of the versions where negative improvement is observed, the effectiveness of ESP degrades less than CBI.

C. Efficiency

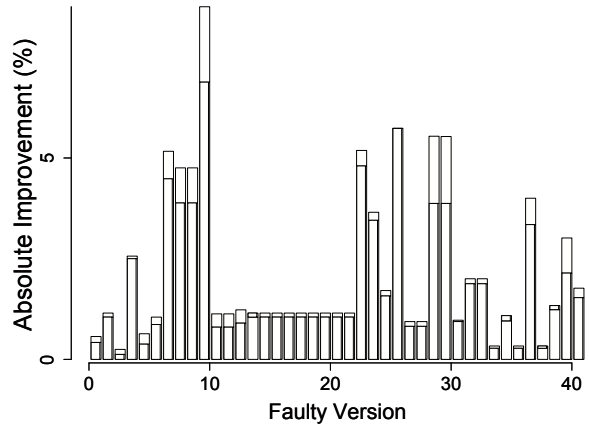
We measured the relative computation time for each of the different versions of the suspiciousness metrics used in our evaluation for each program version. Table IV shows the mean relative efficiency for the reduced bias metrics $\tau^{c}_{ls,p}$ and $\tau^{c,f}_{ls,p}$ when integrated into the *Tarantula*, F_1 and *Ochiai* suspiciousness metrics. As Table IV shows, the algorithms to compute the reduced bias metrics are not inefficient relative to the algorithms that compute the standard suspiciousness metrics. ESP is relatively more efficient than CBI because ESP requires more computation time, which absorbs a portion of the additional computational cost required to solve the regression models for $\tau^{c}_{ls,p}$ and $\tau^{c,f}_{ls,p}$. While ESP incurs ~ 4.5x slowdown compared to CBI, it is more effective [14].

D. Sampling

Predicate-level statistical debuggers such as ESP and CBI use instrumentation to collect predicate data. The collection adds overhead to program execution. The overhead is limited by employing sparse random sampling rather than complete data collection. The sampling collects an unbiased representative set of program behavior across test cases. Here we



(a) Standard *Ochiai* as the reference metric.



(b) *Ochiai* integrated with $\tau^{c}_{ls,p}$ as the reference metric.

Figure 4. *Ochiai* Evaluation.

evaluate $\tau^{c,f}_{ls,p}$ under sparse sampling to determine the extent to which the introduced uncertainty reduces its effectiveness.

Fig. 5 shows the *Cost* incurred by using the F_1 metric integrated with the reduced bias metric $\tau^{c,f}_{ls,p}$ (shaded blocks) within ESP and CBI compared to using the *Importance* metric within ESP and CBI, (non-shaded blocks). The effectiveness of ESP and CBI remains stable under sampling rates of 1/10 and 1/100. For each of these rates, the version of the predicate-level statistical debuggers using the F_1 metric employing $\tau^{c,f}_{ls,p}$ outperforms its counterpart employing the *Importance* metric. For less frequent rates the variance of the *Cost* increases. This is expected given the introduction of random sampling. At a sampling of 1/1,000 the F_1 metric employing $\tau^{c,f}_{ls,p}$ still outperforms the *Importance* metric but the relative difference in effectiveness narrows.

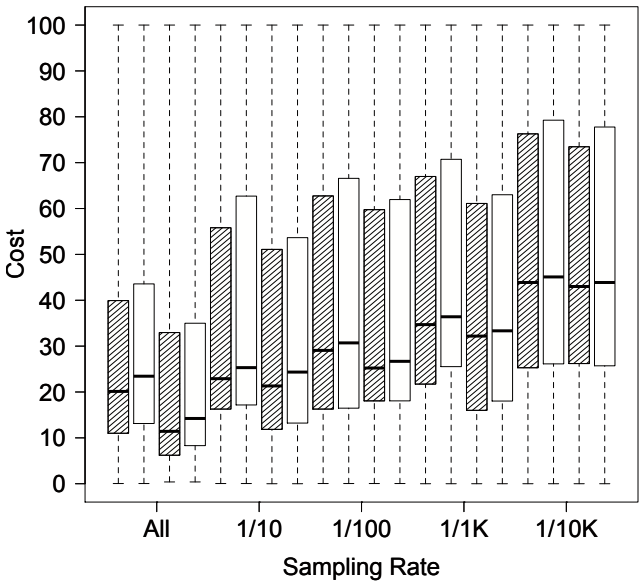


Figure 5. CBI (red) and ESP (white) employing the *Importance* suspiciousness metric (non-shaded) and the F_1 integrated with $\tau^{c,f}_{ls,p}$ (shaded).

The performance of both metrics at a sampling rate of 1/10,000 reveals a trend: sufficiently infrequent rates will reduce the effectiveness of ESP and CBI, regardless of the suspiciousness metric used. However, the performance of ESP and CBI under the more frequent rates shows that $\tau^{c,f}_{ls,p}$ can improve effectiveness of ESP and CBI up to a sampling rate of 1/1,000. This is significant; research has shown that sampling rates $\leq$ 1/1,000 significantly reduce overhead in predicate-level statistical debuggers [1], [2].

E. Validity

Internal, external, and construct validity threats affect our evaluation. Internal validity threats arise when factors affect the dependent variables without evaluators knowledge. It is possible that some implementation flaws could have affected the evaluation results. However, our results for the evaluated benchmarks are similar in magnitude to improvements offered by Baah et al.'s statement-level work [11]. Threats to external validity occur when the results of our evaluation cannot be generalized. Although we performed our evaluations on nine programs with a total of 180 versions and two different predicate-level statistical debuggers (CBI and ESP), we cannot claim that the effectiveness observed in our evaluation can be generalized to other faults in other programs for other predicate-level statistical debuggers. Threats to construct validity concern the appropriateness of the metrics used in our evaluation. More studies into how useful developers find predicate-ranking metrics need to be performed [24]. However, the more accurate fault-localization methods are the more meaningful such studies will become.

V. Related Work

Many debugging approaches use statistical analysis and program coverage data to rank the suspiciousness of program elements [1]–[12], [14]. However, none of these approaches use causal inference to account for *control flow* and *failure flow* confounding biases at the predicate-level.

A related approach is the Probabilistic Program Dependence Graph (PPDG) [9]. The PPDG is a probabilistic model of an entire program, which augments each node of a program-dependence graph with a conditional probability table (CPT) characterizing the conditional-probability distribution of the nodes states, given the states of its parent nodes. Although the technique has been shown to be effective, a node may have an anomalous state without being a cause of a failure. In our approach, we estimate the causal effect of a given predicate being true using a regression model involving only the predicate and its forward control flow predecessor predicate; CPTs are not needed.

State-altering approaches such as Delta Debugging and IVMP attempt to find the cause of program failure by altering program states and re-executing the program [5], [23], [25]. Our approach is more lightweight than these types of approaches. Performing experiments on altered programs can be time consuming and requires an oracle to determine the success or failure of each altered program. Also, previous evaluations suggest that stochastic distributions within subject programs can degrade state-altering analysis [14].

Other debugging approaches use slicing to compute the set of statements that potentially affect the values of a given program point [26], [27]. These techniques do not provide rankings to the developer to facilitate localization. Thus, it is difficult to compare our approach with these approaches.

VI. Conclusion

Recent work has applied causal inference to reduce or eliminate control and data flow dependence confounding bias in statement-level statistical debuggers. Here we further these efforts by applying and extending causal inference to the predicate-level. First we adapted Baah et al.'s statement-level definition of the forward control flow predecessor to the predicate-level. Using the definition we provided a linear regression model, which accounts for *control flow confounding bias* and estimates the effect of a given predicate on a program failure. Next, we extended the model to account for *failure flow confounding bias*. Finally, we presented an evaluation, which showed that the reduced bias metric, $\tau^{c,f}_{ls,p}$, from our extended model significantly improved the effectiveness of existing metrics.

REFERENCES

[1] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," *SIGPLAN Not.*, vol. 38, no. 5, pp. 141–154, May 2003.

[2] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *SIGPLAN Not.*, vol. 40, no. 6, pp. 15–26, Jun. 2005.

[3] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: statistical model-based bug localization," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 286–295, September 2005.

[4] A. X. Zheng, B. Liblit, and M. Naik, "Statistical debugging: simultaneous identification of multiple bugs," in *In ICML 06: Proceedings of the 23rd international conference on Machine learning*. ACM Press, 2006, pp. 1105–1112.

[5] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *Proceedings of the 2008 international symposium on Software testing and analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 167–178.

[6] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282.

[7] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 89–98.

[8] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "Spectrum-based multiple fault localization," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 88–99.

[9] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," *IEEE Transactions on Software Engineering*, vol. 36, pp. 528–545, 2010.

[10] ——, "Causal inference for statistical fault localization," in *International Symposium on Software Testing and Analysis (ISSTA 2010)*, Trento, Italy, July 2010, pp. 73–83.

[11] ——, "Mitigating the confounding effects of program dependences for effective fault localization," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 146–156.

[12] J. A. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proceedings of the 2007 international symposium on Software testing and analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 16–26.

[13] Z. Zhang, W. K. Chan, T. H. Tse, P. Hu, and X. Wang, "Is non-parametric hypothesis testing model robust for statistical fault localization?" *Inf. Softw. Technol.*, vol. 51, no. 11, pp. 1573–1585, Nov. 2009.

[14] R. Gore, P. F. Reynolds, and D. Kamensky, "Statistical debugging with elastic predicates," *Automated Software Engineering, International Conference on*, pp. 492–495, 2011.

[15] S. L. Morgan and C. Winship. Cambridge University Press, Jul. 2007.

[16] J. Pearl, *Causality: Models, Reasoning, and Inference*. Cambridge University Press, Mar. 2000.

[17] G. Casella and R. L. Berger, *Statistical Inference*. Pacific Grove, CA: Wadsworth and Brooks/Cole, 2002.

[18] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987.

[19] "Software-artifact infrastructure repository," http://sir.unl.edu/portal/index.php.

[20] "bc calculator," http://www.gnu.org/software/bc/.

[21] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "Cil: Intermediate language and tools for analysis and transformation of c programs," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02. London, UK: Springer-Verlag, 2002, pp. 213–228.

[22] R Development Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2008, ISBN 3-900051-07-0. [Online]. Available: http://www.R-project.org

[23] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th international conference on Software engineering*, ser. ICSE '05. New York, NY, USA: ACM, 2005, pp. 342–351.

[24] C. Parnin and A. Orso, "Are Automated Debugging Techniques Actually Helping Programmers?" in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2011)*, Toronto, Canada, July 2011, pp. 199–209.

[25] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 272–281.

[26] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 263–272.

[27] F. Tip, "A survey of program slicing techniques," *JOURNAL OF PROGRAMMING LANGUAGES*, vol. 3, pp. 121–189, 1995.