Reducing Confounding Bias in Predicate-level Statistical Debugging Metrics

Ross Gore, Paul F. Reynolds, Jr.
Dept of Computer Science
University of Virginia
Charlottesville, VA, 22901
{rjg7v, pfr}@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 software fault localization [1-21]. 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 program elements. Program elements refer to individual statements, the truth-values of branches or other inserted predicates (conditional propositions). The approaches employ a metric to score the suspiciousness of program elements. Then developers examine program elements in decreasing order of suspiciousness until the fault is discovered.

One suspiciousness metric is the probability of a program \( Q \) failing given that a certain predicate \( p \) is true. This probability, \( \Pr(Q \text{ fails} | p=\text{true}) \), indicates if the condition specified by predicate \( p \) was true during an execution of \( Q \). For predicate-level statistical debuggers \( p=\text{true} \) means an inserted predicate was evaluated to be true.

Given the execution of a test suite, \( \Pr(Q \text{ fails} | p=\text{true}) \) is typically estimated by the sample ratio \( f_p/(f_p+s_p) \), 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 \( \Pr(Q \text{ fails} | p=\text{true}) \) use statistical techniques on observational data to determine the effect of individual predicates on program failures. However, these approaches can be susceptible to biases.

Recently Baah et. al. showed that control flow dependence confounding bias exists within statement-level suspiciousness metrics [14, 15]. 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 [22, 23]. 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 this bias at the statement-level Baah et al. improved the effectiveness for a variety of established statement-level suspiciousness metrics. More recently, Baah et al. improved their approach by incorporating data flow dependences between statements in their definition of the forward control flow predecessor [15].

Here we look to adapt Baah et al.’s work for predicate-level statistical debuggers. The adaptation has challenges requiring innovation. At the predicate-level the number of forward control flow predecessor predicates that are possible depends on the statistical debugging technique employed. This is not true in Baah et al.’s work at the statement-level. One of the major contributions of our work is a robust method to efficiently track forward control flow predecessor predicates and employ them in a causal inference model that eliminates control flow confounding bias in different predicate-level statistical debuggers.

Within predicate-level statistical debugging approaches the issue of bias has been previously explored. Liblit et al. showed that the estimate \( f_p/(f_p+s_p) \) of \( \Pr(Q \text{ fails} | p=\text{true}) \) 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 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. In the Importance metric the suspiciousness of a predicate \( p \) is measured, “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 the
Importance metric, Pr(£ fails | p=true) is estimated by the difference: \( \frac{f_p}{f_p + s_p} - \frac{f_p \text{obs}}{f_p \text{obs} + s_p \text{obs}} \).

Here, \( f_p \text{obs} \) and \( s_p \text{obs} \) 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 major contribution of our work is a causal inference model which accounts for failure flow confounding bias at the predicate-level. The combination of these two contributions yields a predicate-level statistical debugging metric that is significantly more effective than existing metrics because failure and control flow confounding bias are reduced 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. BACKGROUND

A. Predicate-level Statistical Debuggers

Predicate-level statistical debugging approaches represent a class of fault localization techniques that share a common structure. Each approach consists of a set of conditional propositions, or predicates, which are inserted into a program and tested at particular points. A single predicate can be thought of as partitioning the space of all test cases into two subspaces: those satisfying the predicate and those not. Better predicates create partitions that more closely match the space where the fault is expressed. Recall, the predicates are ranked, based on their suspiciousness and guide developers in finding and fixing faults.

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 \( x \): \( x>0 \), \( x=0 \) and \( x<0 \) [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, \( \mu_x \), and standard deviation, \( \sigma_x \), of the values assigned to, or returned from, a variable \( x \). Using these profiled statistics, the CBI predicates are complemented with elastic predicates such as: \( x > (\mu_x + \sigma_x) \) and \( x < (\mu_x - \sigma_x) \) [21].

In ESP and CBI, predicates require two data structures for each executed test case: a one bit feedback report, \( R \), indicating if the test case succeeded or failed and a vector \( V \) with a one bit entry for each predicate. Within \( V \) each entry indicates if the predicate is observed to be true during test case execution. The data for predicate \( p \) from each feedback report \( R \) and each vector \( V \) is aggregated via a suspiciousness metric, which is used to rank the predicate [1, 2].

B. Causal Graphs

Baah et al.’s approach to eliminating dependence confounding bias in statement-level statistical debuggers builds on Pearl's Structural Causal Model [23, 24] and on the “potential outcome” model of Neyman [25] and Rubin [26]. Our goal is to adapt this approach to predicate-level statistical debuggers to eliminate failure flow and control flow confounding bias. Here we review these frameworks.

Causal graphs are used to represent the causal assumptions that permit statistical techniques to be used with observational data and changes. A causal graph is a directed acyclic graph \( G \). Within \( G \), nodes represent random variables and edges represent cause-effect relationships. An edge \( X \rightarrow Y \) indicates that \( X \) causes \( Y \). Each random variable \( X \) has a probability distribution \( P(x) \), which may not be known. The values of random variables are denoted by the corresponding lowercase letter of the random variable.

All causal effects associated with the causal model \( M=\{G,P\} \) can be estimated if \( M \) is Markovian. Markovian means that each random variable \( X_i \) is conditionally independent of all its nondescendants, given the values of its parents (immediately preceding nodes) \( P(A)_i \) in \( G \) [23].

If \( M \) is Markovian then the joint distribution of the random variables is factored as:

\[
p(x_1, x_2, ..., x_n) = \prod p(x_i | pa_i).
\]

Pearl proved that \( M \) will satisfy the Markovian condition if for each node \( X_i \) in \( G \), the relationship between \( X_i \) and its parents can be described by the structural equation [27]:

\[
x_i = f_i(pa_i, u_i).
\]

Thus \( M \) is Markovian if it represents functional relationships (\( f_i \)) among a set of random variables and any external sources of error (\( u_i \)) are mutually independent.

Pearl has shown that Markovian models are a powerful and concise formalism to combine causal estimation together with causal graphs [23]. For a binary cause, there are two states to which each member of the population can be exposed: treatment and control. A treatment is the change an investigator applies to members of the population to assess its effects relative to not applying the change (the control). These states correspond to the values of a causal treatment variable \( T \) where: \( T = 1 \) for treated population members and \( T = 0 \) for controlled population members. Given an outcome variable \( Y \) over the population, there are two potential outcome random variables: \( Y^0 \) and \( Y^1 \). Also, the average treatment effect, \( \tau \), in the population is:

\[
\tau = E[Y^1] - E[Y^0].
\]
However, many problems call for estimating the average treatment effect in a population from a sample $S$. Let $S_A$ be the subset of $S$ consisting of the treatment sample members, and let $S_0$ be the subset of $S$ consisting of control sample members. The estimator of $\tau$, $\hat{\tau}_{re}$, is the difference of the sample means of the outcomes for those in the treatment group ($S_A$) and those in the control ($S_0$) group [22]:

$$\hat{\tau}_{re} = \frac{1}{|S_A|} \sum_{i \in S_A} y_i - \frac{1}{|S_0|} \sum_{i \in S_0} y_i.$$  \hspace{1cm} (4)

In an ideal randomized experiment, sample members are assigned to the treatment group or the control group randomly. Thus the treatment indicator variable $T$ is independent of the potential outcomes $Y^1$ and $Y^0$ and $\hat{\tau}_{re}$ is an unbiased and consistent estimator of $\tau$. Since $\hat{\tau}_{re}$ is unbiased, $E[\hat{\tau}_{re}] = \tau$ and because $\hat{\tau}_{re}$ is consistent it converges to $\tau$ [22]. However, ideal randomized experiments are rare. Instead, sample data often comes from an observational study. In an observational study the effects are not under the control of the investigator and occur in the past. In general, treatment selection is not random, so the treatment indicator variable $T$ is not independent of the potential outcomes $Y^1$ and $Y^0$. Under these circumstances, the estimator $\hat{\tau}_{re}$ is likely to be biased, meaning $E[\hat{\tau}_{re}] \neq \tau$. Thus a better estimate of $\tau$ is needed.

Often one can characterize an estimate of $\tau$ in terms of one or more variables that are suspected of influencing selection. For example, a physician considers the symptoms and medical history of a patient before prescribing treatment. These variables are covariates of the treatment indicator $T$. If a set $X$ of covariates accounts for which members received the treatment and which did not, then the confounding bias when estimating the average treatment effect $\tau$ on an outcome $Y$ can be reduced or eliminated by conditioning on $X$. Pearl’s Back-Door Criterion specifies the required characteristics for such a set of covariates.

However, before presenting the Back-Door Criterion, we must first define blocking and d-separation [23].

**Definition:** A set $S$ of nodes in a causal graph $G$ is said to block a path $p$ if either (1) $p$ contains a chain $U \rightarrow M \rightarrow V$ or a fork $U \leftarrow M \rightarrow V$ whose middle node $M$ is in $S$, or (2) $p$ contains at least one collider $U \leftarrow M \rightarrow V$ such that the middle node $M$ is not in $S$ and no descendant of $M$ is in $S$. If $S$ blocks all paths from $A$ to $B$, $S$ d-separates $A$ and $B$.

If $S$ d-separates $A$ and $B$ then $A$ and $B$ are conditionally independent given $S$, that is, $P(B \mid A, S) = P(B \mid S)$.

**Definition:** A set $S$ of nodes satisfies the Back-Door Criterion relative to a pair of nodes $A$, $B$ in a causal graph $G$ if the following conditions are met:

1. No node in $S$ is a descendant of $A$.
2. $S$ blocks every path between $A$ and $B$ that contains an edge into $A$.

Similarly, if $A$ and $B$ are two disjoint subsets of nodes in $G$, then $S$ satisfies the Back-Door Criterion relative to $A$, $B$ if it satisfies the criterion relative to any pair $A_j \subseteq A$, $B_j \subseteq B$.

The second condition of the Back-Door Criterion is responsible for the criterion’s name. It requires that paths that enter $A$ through the back door be blocked. If $X$ is a set of variables that blocks all back door paths in a causal graph between a treatment indicator $T$ and an outcome variable $Y$, then those back door paths will not contribute to the association between $T$ and $Y$. Hence, by conditioning on $X$, the average treatment effect of $T$ on $Y$ can be estimated without confounding bias [23].

**C. Regression**

There are a number of approaches to using a set $X$ of covariates that satisfy the Back-Door Criterion, relative to a treatment indicator $T$ and an outcome variable $Y$, to construct an unbiased estimator of the average treatment effect $\tau$ [22].

In our work we employ a linear regression model [28]. A generic linear regression model for estimating $\tau$ is:

$$Y = \alpha + \tau T + \beta X + \omega Z + \epsilon.$$  \hspace{1cm} (5)

In this generic model, $\alpha$ is an intercept, $\tau$, $\beta$ and $\omega$ are coefficients of $T$, $X$ and $Z$ respectively, and $\epsilon$ is an error term that is uncorrelated with $T$. The least-squares estimate of $\tau$, denoted by $\hat{\tau}_{ls}$, is unbiased. More complex models have been proposed for causal inference, but we do not explore them due to the lack of fast and robust supporting software.

**III. REDUCING BIASES IN STATISTICAL DEBUGGING**

Here, we demonstrate that confounding and susceptibility bias can be eliminated or reduced from predicate-level statistical debugging metrics. First, we present three established statistical debugging suspiciousness metrics: Tarantula [9], Ochiai [10] and $F_1$ Measure [10]. These metrics can be applied to the canonical predicates employed in CBI and the elastic predicates employed in ESP. Then we describe how control flow confounding bias can be reduced or eliminated from predicate-level suspiciousness metrics by innovatively adapting Baah et al.’s statement-level approach to predicate-level statistical debugging. Finally, we show that the failure flow confounding bias identified in Liblit et al.’s Importance metric [1, 2] can be formalized and made more effective within our approach.

**A. Existing Suspicousness Metrics**

1. **Tarantula**

$$\text{Tarantula} = f_p / \left( (f_p / f) + (s_p / s) \right)$$  \hspace{1cm} (6)
Within Tarantula, \( s_p \) and \( f_p \) are the number of respective tests that succeed and fail where predicate \( p \) is true, and \( s \) and \( f \) are the number of respective tests that succeed and fail [9]. When \( s = f \) the Tarantula formula simplifies to our original estimate for the probability of failure given that predicate \( p \) is true, \( f_p / (f_p + s_p) \) [14].

2) \( F_1 \) Measure and Ochiai

\[ Ochiai = \sqrt{\left( f_p / f \right) * \left( f_p / (f_p + s_p) \right)} \]  

\[ F_1 \text{Measure} = \frac{2}{\left( \frac{1}{f_p / f} \right) + \left( \frac{1}{f_p / (s_p + f_p)} \right)} \]  

Both sensitivity (or recall) and specificity (or precision) are estimated within the Ochiai metric and the \( F_1 \) Measure [10]. Sensitivity is the probability of failure given that predicate \( p \) is true, \( \text{Pr}(Q \text{ fails} | \ p = \text{true}) \), and is estimated by \( f_p / (f_p + s_p) \). Specificity is the probability that predicate \( p \) is true given failure \( \text{Pr}(p = \text{true} | Q \text{ fails}) \), and is estimated by \( f_p / f \). The Tarantula metric only includes an estimate of specificity [14]. However, sensitivity is also important. A predicate \( p \) for which \( \text{Pr}(p = \text{true} | Q \text{ fails}) \) is very low will be true in very few failing test cases. If there are many passing test cases where \( p \) is true, the overall sample of test cases where \( p \) is true will be unbalanced. If there are few passing test cases where \( p \) is true, the overall sample of test cases where \( p \) is true will be small, even if it is not unbalanced. The use of sensitivity in these metrics addresses this issue.

In information retrieval, estimates of sensitivity and specificity are balanced using the harmonic mean [1, 10]. In biology, sensitivity and specificity are balanced via the geometric mean [10]. The Ochiai formula computes the geometric mean of the estimated sensitivity and specificity while the \( F_1 \) Measure computes the harmonic mean.

Llibit et al.’s Importance metric is closely related to the \( F_1 \) Measure. For a given predicate \( p \), Importance\((p)\) estimates specific with the quantity \( \text{Increase}(p) = \text{Fail}(p) - \text{Context}(p) \). Within \( \text{Increase}(p) \), \( \text{Fail}(p) = f_p / (f_p + s_p) \) and \( \text{Context}(p) = f_{p, \text{obs}} / (f_{p, \text{obs}} + s_{p, \text{obs}}) \). Recall, \( f_{p, \text{obs}} \) and \( s_{p, \text{obs}} \) are the number of respective failing and succeeding test cases where predicate \( p \) is reached and evaluated (true or false) [1, 2].

\( \text{Fail}(p) \) is identical to the specificity estimate of \( \text{Pr}(Q \text{ fails} | \ p = \text{true}) \) in the other metrics. However, the second term, \( \text{Context}(p) \), is a correction of the failure flow confounding bias that can occur when estimating specificity. \( \text{Context}(p) \) ensures that predicate \( p \) is scored “not by the chance that it implies failure, but by how much difference it makes that the predicate is observed to be true versus simply reaching the line where the predicate is evaluated” [1].

Traditionally, the sensitivity estimate, \( f_p / f \), within Importance\((p)\) is given a logarithmic transformation, \( \log(f_p) / \log(f) \). This transformation moderates the impact of large numbers of failures and has been shown to improve effectiveness [1, 2]. However, in order to enhance comparability to the other metrics and existing work we do not employ the logarithmic transformation here. Thus, in our work the Importance metric is:

\[ \text{Importance} = \frac{2}{\frac{1}{f_p / f} + \text{Increase}(p)} \]  

Next, we innovatively adapt Baah et. al’s work [14] to reduce control flow confounding bias at the predicate-level.

B. Eliminating Control Flow Confounding Bias

The adaptation of Baah et. al’s work [14] from the statement-level to the predicate-level has challenges. Recall, at the predicate-level, the number of possible forward control flow predecessor predicates depends on the debugging approach employed. This is not true at the statement-level. ESP can use as many as 12 predicates for each instrumented program point resulting in 12 different possible forward control flow predecessor predicates while CBI uses 3 predicates resulting in 3 different possible forward control flow predecessor predicates [1, 2, 21]. Addressing this challenge requires innovation. First, we define forward control flow predecessor predicates. Then we propose a linear regression model for reducing or eliminating control flow confounding bias in the specificity estimate within predicate-level suspiciousness metrics.

1) Forward Control Flow Predecessor Predicates

A program’s dependence graph is a directed graph whose nodes correspond to program statements and whose edges represent data and control dependences between statements [29]. 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 \( X \) [29]. 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 a 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 \( Y \).
Defining forward control flow predecessor predicates, as opposed to forward control flow predecessor statements, requires an additional step. At each assignment to, or return of, a variable \( x \), all variables \( y_1, y_2, \ldots, y_n \) referenced in the forward control flow predecessor statement are identified. Each pair of variables \( (x, y_i) \) induces additional predicate instrumentation that partitions the value of \( x \) and \( y_i \) with compound predicates. Thus for a given test case and predicate \( p \), the forward control flow predecessor predicate is the set of predicates that correspond to the control flow predecessor statement for \( p \) and are true when combined with \( p \) via a compound predicate. For CBI, the nine compound predicates for each \( (x, y_i) \) are shown in Table 1. The compound predicates employed in ESP are formed in the same manner. While we did not create compound predicates [17], their use to capture forward control flow predecessor predicates is our innovation.

TABLE I. COMPOUND CBI PREDICATES FOR \((X, Y_i)\)

<table>
<thead>
<tr>
<th>( y_i )</th>
<th>( x = 0 )</th>
<th>( x &lt; 0 )</th>
<th>( x &gt; 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_i = 0 )</td>
<td>( x = 0 \wedge y_i = 0 )</td>
<td>( x &lt; 0 \wedge y_i = 0 )</td>
<td>( x &gt; 0 \wedge y_i = 0 )</td>
</tr>
<tr>
<td>( y_i &lt; 0 )</td>
<td>( x = 0 \wedge y_i &lt; 0 )</td>
<td>( x &lt; 0 \wedge y_i &lt; 0 )</td>
<td>( x &gt; 0 \wedge y_i &lt; 0 )</td>
</tr>
<tr>
<td>( y_i &gt; 0 )</td>
<td>( x = 0 \wedge y_i &gt; 0 )</td>
<td>( x &lt; 0 \wedge y_i &gt; 0 )</td>
<td>( x &gt; 0 \wedge y_i &gt; 0 )</td>
</tr>
</tbody>
</table>

Implementing forward control flow predecessor predicates requires: (1) the identification of the forward control flow predecessor statement and (2) instrumentation for the compound predicates. It is important to note that the compound predicate instrumentation does not occur for all local or global variables that are in the scope of a variable in a given predicate \( p \). Instead, it only occurs for those variables in the forward control flow predecessor statement of \( p \). This allows our approach to remain efficient relative to existing implementations of CBI and ESP.

2) Linear Regression Model

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 adapted from Baah et al. reduces or eliminates control flow confounding bias [14]:

1. For each predicate \( p \) in a faulty program \( Q \), fit a separate linear model \( M_p \) according to the following:
   a. The outcome variable \( Y \) is 1 for a test if it fails and 0 otherwise.
   b. The treatment indicator \( T_p \) is 1 for a test if \( p \) is true and 0 otherwise.
   c. If \( p \) has a forward control flow predecessor predicate(s) \( cfp(p) \), then \( M_p \) has a single binary covariate \( C_p \), which is 1 for a test if it covers \( cfp(p) \) and is 0 otherwise. If \( p \) does not have a forward control flow predecessor predicate then \( M_p \) has no covariates.

2. Rank predicates in descending order of \( \tau'_p \) - the least-squares estimate, for each predicate, of the coefficient of \( T_p \) in \( M_p \). \( \tau'_p \) is the specificity estimate used in the suspiciousness metric for each predicate. The resulting linear model for predicate \( p \) is:

\[
Y = \alpha_p + \tau_p T_p + \beta_p C_p + \epsilon_p.
\]  

The coefficient \( \tau_p \) is the average treatment effect on test outcomes where predicate \( p \) is true. This is an estimate of specificity, \( \Pr(Q \text{ fails } | p = \text{true}) \) [14].

Recall, the model in (10) includes a set \( X \) of covariates that block all back-door paths in the causal graph between the treatment indicator \( T \) and the outcome variable \( Y \). Thus, for any program where failure is determined with a single output statement and data 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(s) of \( p \), \( cfp(p) \). As a result, all back-door paths to \( p \) are blocked, reducing or eliminating the control flow confounding bias in, \( \tau'_p \), our estimate of the average treatment effect, \( \tau_p \).

C. Eliminating Failure Flow Confounding Bias

Recall, Liblit et. al’s Importance metric is very similar to the \( F_1 \) Measure. Both metrics balance estimates of sensitivity and specificity via the harmonic mean. The difference between them lies in the estimate of specificity. In the Importance metric, \( \text{Increase}(p) \) is used to estimate specificity while \( f_p / (f_p + s_p) \) is used to estimate specificity in the standard \( F_1 \) Measure. Within \( \text{Increase}(p) \), the Context\((p) \) term is used as a heuristic to reduce failure flow confounding bias. While the Context\((p) \) term 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 via a causal model requires ensuring that the set of covariates within the model satisfy the Back-Door Criterion. The Context\((p) \) term in the Importance metric does not meet these requirements, in part, because it does not include any notion of control flow dependence. In this subsection we will demonstrate how the linear model shown in (10) can be improved to reduce failure flow confounding bias.

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 and failure flow confounding biases:

1. For each predicate \( p \) in a faulty program \( Q \), fit a separate linear model \( M'_p \) according to the following:
a. The outcome variable $Y^t$ is 1 for a test if it fails and is 0 otherwise.

b. The treatment indicator $T^t_p$ is 1 for a test if $p$ is true and is 0 otherwise.

c. If $p$ has a forward control flow predecessor predicate(s) $cfp(p)$, then $M^t_p$ has a binary covariate, $C^t_p$ which is 1 for a test if $cfp(p)$ is true and is 0 otherwise. If $p$ does not have a forward control flow predecessor predicate then $M^t_p$ does not have a covariate $C^t_p$.

d. If $p$ is evaluated then $M^t_p$ has a covariate $D^t_p$ for a test, which is 1 and 0 if $p$ is not evaluated.

2. Rank predicates in descending order of $\tau_{ls,p}''$ - the least-squares estimate, for each predicate, of the coefficient of $T^t_p$ in $M^t_p$. $\tau_{ls,p}''$ is the specificity estimate used in the suspiciousness metric for each predicate. The resulting linear model for predicate $p$ is:

$$Y^t = \alpha^t_p + \tau_{ls,p}'' T^t_p + \beta_p C^t_p + \omega_p' D^t_p + \varepsilon^t_p .$$

Once again, the coefficient $\tau_{ls,p}''$ is the average treatment effect on test outcomes where $p$ is true. Also, the forward control flow predecessor, $cfp(p)$, blocks off all Back-Door paths to $p$ ensuring that $C^t_p$ and $D^t_p$ satisfy the Back-Door Criterion. The notable difference between (11) and (10) is the inclusion of the covariate $D^t_p$, which reflects whether or not a predicate was evaluated. The goal of the covariate $D^t_p$ is similar to that of $Context(p)$ within the Importance metric. $D^t_p$ accounts for those instances where the predicate is observed to be true versus simply reaching the line where the predicate is evaluated [1, 2]. Without $D^t_p$ once the fault in a program has been triggered the program flow biases the estimate because the observations collected from subsequent predicates are more susceptible to failure. The result is the specificity estimate, $\tau_{ls,p}''$, which can be employed alone or integrated into existing suspiciousness metrics in the predicate-level statistical debuggers such as CBI and ESP.

D. Integrating $\tau_{ls,p}''$ into Existing Suspicousness Metrics

The specificity estimates $\tau_{ls,p}$ and $\tau_{ls,p}''$ are closely related to the metrics we presented in Section 3. Here, we review how they can be integrated into the Tarantula, $F_i$ Measure, and $Ochiai$ metrics. Recall, when there are an equal number of passing and failing test cases, the Tarantula metric is the standard estimate for specificity [14]. Thus, replacing the Tarantula metric with our reduced bias specificity estimates, $\tau_{ls,p}$ and $\tau_{ls,p}''$ is a straightforward substitution.

However, within the $F_i$ Measure and the $Ochiai$ metric the integration is slightly more complex. Recall, these metrics balance an estimate of specificity with an estimate of sensitivity. We must convert each reduced bias specificity estimate into a probability value before it is substituted for its biased counterpart. This conversion ensures each specificity estimate is within the same range as the sensitivity estimate, (0.0-1.0), and the combination of the two estimates is meaningful. We use the inverse logit function, $\text{invlogit}(x) = \exp(x)/(1+\exp(x))$ to convert our reduced bias specificity estimates [30]. Once the estimate is converted it can be substituted into the $F_i$ Measure and $Ochiai$ metric in place of the standard specificity estimate.

IV. Evaluation

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 2. Each was obtained from [31], except bc, which was obtained from [32].

<table>
<thead>
<tr>
<th>Name</th>
<th>LOC</th>
<th>Vers.</th>
<th>Tests</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tcas</td>
<td>138</td>
<td>41</td>
<td>1608</td>
<td>altitude separation</td>
</tr>
<tr>
<td>totinfo</td>
<td>396</td>
<td>23</td>
<td>1052</td>
<td>information measure</td>
</tr>
<tr>
<td>schedule</td>
<td>299</td>
<td>9</td>
<td>2650</td>
<td>priority queue</td>
</tr>
<tr>
<td>schedule2</td>
<td>297</td>
<td>9</td>
<td>2710</td>
<td>priority queue</td>
</tr>
<tr>
<td>print-tokens</td>
<td>472</td>
<td>5</td>
<td>4130</td>
<td>lexical analyzer</td>
</tr>
<tr>
<td>print-tokens2</td>
<td>399</td>
<td>10</td>
<td>4115</td>
<td>lexical analyzer</td>
</tr>
<tr>
<td>replace</td>
<td>512</td>
<td>31</td>
<td>5542</td>
<td>pattern recognition</td>
</tr>
<tr>
<td>sed</td>
<td>6,092</td>
<td>7</td>
<td>363</td>
<td>stream editing utility</td>
</tr>
<tr>
<td>space</td>
<td>14,382</td>
<td>35</td>
<td>157</td>
<td>ADL interpreter</td>
</tr>
<tr>
<td>bc (1.06)</td>
<td>14,288</td>
<td>1</td>
<td>4,000</td>
<td>basic calculator</td>
</tr>
<tr>
<td>gzip</td>
<td>7,266</td>
<td>9</td>
<td>217</td>
<td>compression utility</td>
</tr>
</tbody>
</table>

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* (v1.06) is a calculator program with a reported buffer overflow fault [4]. 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 [31].

In total, we evaluated 180 faulty versions. We instrumented each version to enable construction of the dynamic control-flow graph for each function. Then we computed the dynamic control-dependence graph for each function from its dynamic control-flow graph. The control-dependence graphs and program instrumentation use the CIL framework [33], which supports C program analysis.

For each test case we computed feedback reports to report success or failure and predicate-truth and predicate-evaluation vectors to report which predicates were true and evaluated respectively. Our approach uses the vectors, feedback reports and control-dependence graph as inputs. The linear regression models and the suspiciousness metrics are implemented in the statistical language R [34].

### B. Ranking Effectiveness

To measure the effectiveness of the suspiciousness metrics we use an established cost-measuring function (*Cost*) [1-3, 5-11, 13-21, 35]. *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* 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 specificity estimates, \( \tau'_{ls,p} \) and \( \tau''_{ls,p} \), into the *Tarantula*, *F*<sub>1</sub> *Measure* and *Ochiai* metrics. We evaluate each reduced bias metric in CBI (red) and ESP (blue). 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 improvement of the metric for the matching debugger. The total height of both portions reflects the improvement of the metric for the debugger matching the colored portion of the bar furthest from the x-axis. The effectiveness of CBI relative to ESP is discussed later.

While our evaluation is in terms of predicates, some statistical debugging evaluations are in terms of statements. Here we note that converting a ranked list of predicates to a ranked list of statements is a straightforward process. First, for each statement identify the corresponding predicate with the highest suspiciousness score and move the statement and the suspiciousness score to set *ST*. Then place the statements in *ST* in descending order by suspiciousness score.

#### 1) Predicate-level Tarantula Suspiciousness Metric

Here, we evaluate the effectiveness of the two reduced bias specificity estimates \( \tau'_{ls,p} \) and \( \tau''_{ls,p} \), compared to the standard *Tarantula* suspiciousness metric in the predicate-level statistical debuggers CBI and ESP. \( \tau'_{ls,p} \) and \( \tau''_{ls,p} \) are obtained from the linear models presented in (10) and (11). Fig. 1 shows that the reduced control flow confounding bias estimate, \( \tau'_{ls,p} \), performs better than the standard *Tarantula* metric on 94 program versions within CBI and ESP but it performs worse on two versions. \( \tau''_{ls,p} \) performs the same as the original *Tarantula* metric on 84 versions. Fig. 2 compares \( \tau'_{ls,p} \) and \( \tau''_{ls,p} \). It uses \( \tau'_{ls,p} \) as the reference metric and measures the Cost of \( \tau''_{ls,p} \) subtracted from the Cost of \( \tau'_{ls,p} \). \( \tau''_{ls,p} \) performs better than \( \tau'_{ls,p} \) for 68 of the 180 versions and it never performs worse.
Fig 3: Results of integrating $\tau_{ls,p}$ into the $F_1$ Measure.

Fig 4: Results using $\tau''_{ls,p}$ within the $F_1$ Measure metric and using $\lambda''_{ls,p}$ within the $F_1$ Measure as the reference metric.

Fig 5: Results of integrating $\tau'_{ls,p}$ within the Ochiai metric.

Fig 6: Results using $\tau''_{ls,p}$ within the Ochiai metric and $\tau'_{ls,p}$ within the Ochiai as the reference metric.

1) Predicate-level $F_1$ Measure Suspiciousness Metric
Here, we evaluate the effectiveness of $\tau_{ls,p}$ and $\tau''_{ls,p}$ within the $F_1$ Measure in CBI and ESP. Fig. 3 shows that the $F_1$ Measure employing $\tau_{ls,p}$ performs better than the $F_1$ Measure using the standard specificity estimate on 92 versions within CBI and ESP but performs worse on seven versions. The metrics perform the same on 81 versions. Fig. 4 compares the version of the $F_1$ Measure employing $\tau_{ls,p}$ to the version employing $\tau''_{ls,p}$. Again, the metric using $\tau''_{ls,p}$ outperforms the metric using $\tau'_{ls,p}$. The $F_1$ Measure employing $\tau''_{ls,p}$ performs better than the $F_1$ Measure employing $\tau'_{ls,p}$ for 54 of the 180 programs and it never performs worse.

2) Predicate-level Ochiai Metric
Here, we evaluate the effectiveness of $\tau'_{ls,p}$ and $\tau''_{ls,p}$ within the Ochiai metric in CBI and ESP. Fig. 5 shows that the Ochiai metric employing $\tau'_{ls,p}$ performs better than the Ochiai metric using the standard specificity estimate on 59 versions within CBI and ESP, but performs worse on only two versions. The metrics perform the same on 119 versions. Fig. 6 compares the version of the Ochiai metric employing $\tau'_{ls,p}$ to the version employing $\tau''_{ls,p}$. Once again, the metric employing $\tau''_{ls,p}$ outperforms the metric using $\tau'_{ls,p}$. The Ochiai metric employing $\tau''_{ls,p}$ performs better than the Ochiai metric employing $\tau'_{ls,p}$ for 41 of the 180 programs and it never performs worse.

3) Discussion
In each portion of the evaluation, the suspiciousness metric employing the specificity estimate $\tau''_{ls,p}$ is the most effective metric within both CBI and ESP. These results show that failure and control flow confounding bias exist in established predicate-level statistical debugging suspiciousness metrics and that our specificity estimate, $\tau''_{ls,p}$, reduces or eliminates these confounding biases.
Space precludes a graphical comparison of $\tau''_{b,p}$ with the Importance metric. However, we have evaluated the Importance metric against the reduced bias versions of Tarantula, $F_1$ Measure and Ochiai metrics employing $\tau''_{b,p}$. Table 3 shows that in the preponderance of the program versions the suspiciousness metric employing $\tau''_{b,p}$ outperforms the Importance metric.

<table>
<thead>
<tr>
<th>Suspiciousness Metric</th>
<th>Better than Importance</th>
<th>Same as Importance</th>
<th>Worse than Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tarantula</td>
<td>102</td>
<td>30</td>
<td>48</td>
</tr>
<tr>
<td>$F_1$ Measure</td>
<td>110</td>
<td>41</td>
<td>29</td>
</tr>
<tr>
<td>Ochiai</td>
<td>114</td>
<td>38</td>
<td>28</td>
</tr>
</tbody>
</table>

Furthermore, in subsection D the $F_1$ Measure employing $\tau''_{b,p}$ is evaluated favorably against the Importance metric at different predicate sampling rates. Thus for CBI and ESP and the evaluated programs, suspiciousness metrics employing $\tau''_{b,p}$ are superior to the Importance metric.

Although the specificity estimates $\tau'_{b,p}$ and $\tau''_{b,p}$ 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 predicate $p$ will necessarily trigger a failure, if the statement is faulty [14]. However, covering a faulty statement corresponding to predicate $p$ may not be sufficient to trigger a failure because either the statement does not cause an invalid internal state or an invalid internal state does not propagate to the program’s 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 $\tau_{b,p}$ and $\tau''_{b,p}$ 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 [21]. While $\tau'_{b,p}$ and $\tau''_{b,p}$ 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 4 shows the mean relative efficiency for the reduced bias specificity estimates $\tau'_{b,p}$ and $\tau''_{b,p}$ when integrated into the Tarantula, $F_1$ Measure and Ochiai suspiciousness metrics. As Table 4 shows, the algorithms to compute the reduced bias specificity estimates are efficient 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 linear regression models for $\tau'_{b,p}$ and $\tau''_{b,p}$. While ESP incurs ~4.5x slowdown compared to CBI, it has been shown to be more effective [21].

<table>
<thead>
<tr>
<th>Suspiciousness Metric</th>
<th>Standard (CBLESP)</th>
<th>$\tau'_{b,p}$ (CBLESP)</th>
<th>$\tau''_{b,p}$ (CBLESP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tarantula</td>
<td>1.00, 1.00</td>
<td>1.94, 1.24</td>
<td>2.16, 1.36</td>
</tr>
<tr>
<td>$F_1$</td>
<td>1.05, 1.02</td>
<td>1.97, 1.25</td>
<td>2.18, 1.37</td>
</tr>
<tr>
<td>Ochiai</td>
<td>1.06, 1.04</td>
<td>1.99, 1.25</td>
<td>2.18, 1.37</td>
</tr>
</tbody>
</table>

### D. Sampling

Predicate-level statistical debuggers such as ESP and CBI use source code 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 evaluate $\tau''_{b,p}$ under sparse random sampling to determine the extent to which the uncertainty introduced by sampling reduces its effectiveness.

The tcas benchmark is not included in this portion of the evaluation because it contains no loops, is less than 200 lines of code and can be executed in less than five seconds. Sampling is not needed for tcas and including the results would not be useful.

Fig. 7 shows the Cost incurred by using the $F_1$ Measure with the integrated reduced bias specificity estimate $\tau''_{o,p}$ (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 $F_1$ Measure employing $\tau''_{o,p}$ 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 rate of 1/1/000 the $F_1$ Measure employing $\tau''_{o,p}$...
still outperforms the Importance metric but the relative difference in effectiveness narrows. 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 specificity estimate used in the metric. However, the performance of ESP and CBI under the more frequent rates shows that $\tau_{l,p}$ 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 [14]. 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. However, the more accurate fault-localization methods are the more meaningful such studies will become.

V. Related Work

Many debugging approaches [1-3, 5-18, 21] use statistical analysis and program coverage data to rank the suspiciousness of program elements. However, none of these approaches use causal inference to account for control flow and failure flow confounding biases. This is the difference between our approach and existing statistical approaches.

A related approach is the Probabilistic Program Dependence Graph (PPDG) [12]. 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 node’s 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 linear regression model involving only the predicate and its forward control flow predecessor predicates; CPTs are not needed. Furthermore PPDG cannot address failure flow bias.

State-altering approaches such as Delta Debugging and IVMP [7, 8, 35, 36] attempt to find the cause of program failure by altering program states and re-executing the program. 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 [21].

Other debugging approaches use slicing [37, 38, 39] to compute the set of statements that potentially affect the values of a given program point. These techniques differ from most fault localization techniques, including ours, because they do provide any guidance, such as rankings, to the developer to facilitate the localization process. 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 innovatively applying and extending causal-inference to the predicate-level. First we innovatively 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 biases 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 specificity estimate from our extended model significantly improved the effectiveness of existing suspiciousness metrics. In future work our model will account for test suite composition to improve the reliability of estimates in the face of poor coverage data and we will expand the definition of forward control flow predecessor predicates to include dynamic data dependencies.
REFERENCES


