Behavioral Fault Localization by Sampling Suspicious Dynamic Control Flow Subgraphs

Tim A. D. Henderson and Andy Podgurski
Dept. of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, Ohio, USA 44106
tadh@case.edu, podgurski@case.edu

***Abstract*—We present a new algorithm, Score Weighted Random Walks (SWRW), for behavioral fault localization. Behavioral fault localization localizes faults (bugs) in programs to a group of interacting program elements such as basic blocks or functions. SWRW samples suspicious (or discriminative) subgraphs from basic-block level dynamic control flow graphs collected during the execution of passing and failing tests. The suspiciousness of a subgraph may be measured by any one of a family of new metrics adapted from probabilistic formulations of existing coverage-based statistical fault localization metrics. We conducted an empirical evaluation of SWRW with nine subgraph-suspiciousness measures on five real-world subject programs. The results indicate that SWRW outperforms previous fault localization techniques based on discriminative subgraph mining.**

I. Introduction

Automated fault localization techniques have been developed to help programmers locate software faults (bugs) responsible for observed software failures. Many of these techniques are statistical in nature (e.g., [1]–[3]). They employ statistical measures of the *association*, if any, between the occurrence of failures and the execution of particular program elements like statements or conditional branches. The program elements that are most strongly associated with failures are identified as "suspicious", so that developers can examine them to see if they are faulty. The association measures that are used are often called *suspiciousness metrics* [4]. Such *statistical fault localization* (SFL) techniques typically require *execution profiles* (or *spectra*) and PASS/FAIL labels for a set of both passing and failing program runs. Each profile entry characterizes the execution of a particular program element during a run. For example, a statement-coverage profile for a run indicates which statements were executed at least once. The profiles are collected with program instrumentation, while the labels are typically supplied by software testers or end users.

Kochhar *et al.* [5] recently surveyed 386 software engineering practitioners about their expectations for automated fault localization. While practitioners indicated their preference for very accurate algorithms, over 85% of respondents also indicated their preference for tools which help them understand the output of fault localization algorithms. This is an important finding as most statistical approaches do not provide an explanation of their results. The SFL techniques often simply compute suspiciousness measures and rank the program elements accordingly. These rankings may be helpful, but without more information programmers could overlook the faulty element even when it is ranked highly.

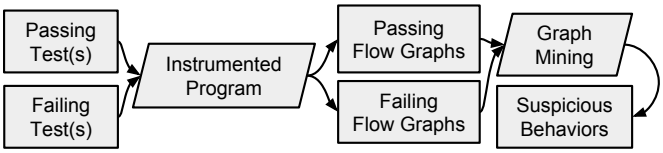


Fig. 1: Process for localizing faults with discriminative graph mining.

Suspicious-Behavior Based Fault Localization (SBBFL) is a statistical fault localization technique that aids the programmer in understanding suspiciousness scores by providing a *context* of interacting elements.[1] Instead of implicating a single element, SBBFL implicates a larger runtime behavior (see process in Figure 1). The implicated control flow paths (or subgraphs) may help the programmer understand the nature of a bug [7].

We present a new algorithm, *Score-Weighted Random Walks* (SWRW), for behavioral fault-localization. SWRW belongs to a family of *discriminative graph-mining algorithms* that have previously been used for behavioral fault localization [3], [7]–[15]. Graph mining is very powerful in principle but algorithms must make trade-offs to address the challenging combinatorics of the graph mining problem. Our new algorithm, SWRW, mitigates the combinatorics by randomly sampling "suspicious" subgraphs from dynamic control flow graphs. During the sampling process, the most suspicious subgraphs (as judged by a suitable suspiciousness metric) are favored for selection. Unlike previous algorithms, SWRW can be used with a wide variety of suspiciousness metrics — which allows it to use better metrics than available to previous work. Even when using the same metric as similar algorithms, SWRW localizes faults more accurately than they do.

Summary of Contributions

1) A new behavioral fault localization algorithm, SWRW, that samples suspicious subgraphs from dynamic control flow graphs. Unlike similar algorithms, SWRW can be used with a variety of suspiciousness metrics.
2) New generalizations of existing suspiciousness metrics that allow them to be applied to behaviors represented by subgraphs of dynamic control flow graphs.

[1]Dynamic slicing [6] also provides such a context, but does not in itself involve suspiciousness measures.

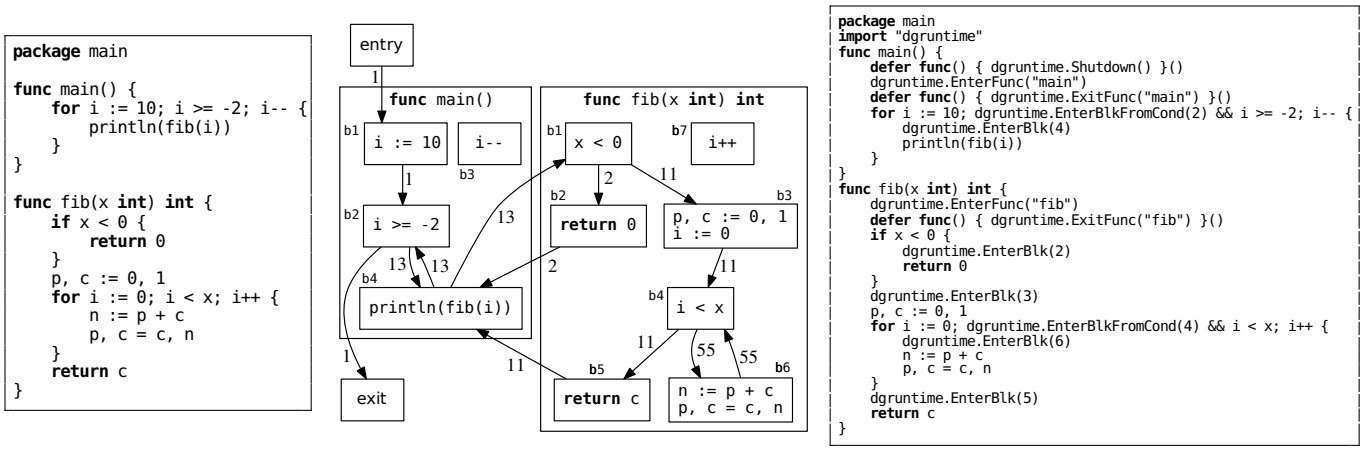


Fig. 2: An example dynamic control flow graph (DCFG) for a Go program (listing on left) that computes elements from the Fibonacci sequence. Each vertex is a basic block with a basic block identifer (e.g. `b1`) that, in conjunction with the name of the containing function, serves as the label for the block (e.g. `main:b1`). Each edge shows the number of traversals taken during the execution of the program. Note that the loop update blocks (`main:b3` and `fib:b7`) will not be in the profiles because Dynagrok instruments the Go source code and profiling instructions cannot be syntactically inserted in those locations. The instrumented program is shown on the right.

3) An empirical study whose results suggest that SWRW is more accurate than similar algorithms.
4) Dynagrok, a new instrumentation, mutation, and analysis tool for the Go programming language.

II. Dynagrok: A New Profiling Tool

All *Coverage-Based Statistical Fault Localization* (CBSFL) techniques use *coverage profiles* to gather information on how software behaved when executed on a set of test inputs. A coverage profile typically contains an entry for each program element of a given kind (e.g., statement, basic block, branch, or function), which records whether (and possibly how many times) the element was executed during the corresponding program run. The profiles and PASS/FAIL labels for all tests are then used to compute a statistical *suspiciousness score* for each program element.

The process of gathering the coverage information from running programs is called *profiling* and there are many different varieties of profilers and profiling techniques available. Coverage profiling is a simple and widely implemented technique, which is why it has been widely used by the fault localization community. Another technique is *tracing*, which logs the sequence of program locations as they are executed. The traces provide detailed information on the behavior of the program but could grow to be very large for long running programs. This paper uses *execution flow profiling* which computes the dynamic interprocedural control flow graph of a program's execution. This provides some of the benefits of tracing without recording an excessive amount of data.

To capture execution flow profiles we developed *Dynagrok*, a new analysis, instrumentation and mutation platform for the Go programming language.[2] Go is a newer language (2009) from Google that has been seeing increasing adoption in industry. It has been adopted for web programming, systems programming, "DevOps," network programing, and databases.[3] Dynagrok builds upon the *abstract syntax tree* (AST) representation provided by the Go standard library.

Dynagrok collects profiles by inserting instrumentation into the AST of the subject program. The profiles currently collected are *dynamic control flow graphs* (DCFGs) whose vertices represent basic blocks. A *basic block* is a sequence of program operations that can only be entered at the start of the sequence and can only be exited after the last operation in the sequence [16]. A basic-block level *control flow graph* (CFG) is a directed labeled graph $g = (V, E, l)$ comprised of a finite set of vertices V, a set of edges $E \subseteq V \times V$, and a labeling function l mapping vertices and edges to labels. Each vertex $v \in V$ represents a basic block of the program. Each edge $(u, v) \in E$ represents a transition in program execution from block u to block v. The labeling function l labels the basic blocks with a unique identifier (e.g. `function-name:block-id`), which is consistently applied across multiple executions but is never repeated in the same execution.

Figure 2 shows an example DCFG collected by Dynagrok for a simple program that computes terms of the Fibonacci sequence. To collect such graphs Dynagrok parses the program into an AST using Go's standard library. Dynagrok then uses a custom control flow analysis to build static control flow graphs. Each basic block holds pointers to the statements inside of the AST. The blocks also have a pointer to the enclosing *lexical block* in the AST. Using this information, Dynagrok inserts profiling instructions into the AST at the beginning of each basic block. The instructions inserted by Dynagrok use its `dgruntime` library to track the control flow of each thread (which is called a *goroutine* in Go). When the program shuts down (either normally or abnormally) the `dgruntime` library merges the flow graphs from all the threads together and writes out the result.

[2]https://github.com/timtadh/dynagrok

[3]tiobe.com/tiobe-index/, blog.golang.org/survey2016-results

III. From Suspicious Locations to Suspicious Behaviors

Before explaining our notion of a suspicious behavior we will review the concept of suspicious statements, blocks, or other locations in a program. There is a large body of work on coverage based statistical (also called spectrum-based) fault localization (CBSFL) (e.g., [1], [17]–[26]), which identifies suspicious program locations from code coverage profiles collected from passing and failing program runs. All of this work tries to assess the suspiciousness of a particular program element based on a statistical measure of the association between coverage of the element and the occurrence of program failure.

A simple measure of the suspiciousness of a program location l is the probability $\Pr[F|l]$ that the program will fail given execution of the location l [26]. Let n be the total number of executions, f be the number of executions that failed, p be the number of executions that did not fail, n_l be the total number of times l was executed, f_l be the total number of times l was executed and the program failed, and p_l be the total number of times l was executed and the program did not fail. Then we can define an estimator for $\Pr[F|l]$ in terms of these counts by:

$$\Pr[F|l] = \frac{\Pr[F \cap l]}{\Pr[l]} \approx \frac{\frac{f_l}{n}}{\frac{n_l}{n}} = \frac{f_l}{n_l} \tag{1}$$

Other measures have been developed to assess the suspiciousness of program locations but most can be expressed using combinations of simple estimators of just four probabilities: the probability of the program failing $\Pr[F] \approx \frac{f}{n}$, the probability of the program not failing (test passing) $\Pr[P] \approx \frac{p}{n}$, the probability of the execution of a location l when the program fails $\Pr[F \cap l] \approx \frac{f_l}{n}$, and the probability of the execution of a location l when the program does not fail $\Pr[P \cap l] \approx \frac{p_l}{n}$. For example, using these estimators the popular *Ochiai* metric [18] can be expressed [20] as:

$$\begin{aligned} Och &\approx \sqrt{\Pr[F|l] \times \Pr[l|F]} \\ &= \sqrt{\frac{\Pr[F \cap l]}{\Pr[F \cap l] + \Pr[P \cap l]} \times \frac{\Pr[F \cap l]}{\Pr[F]}} \end{aligned} \tag{2}$$

Each simple program statement or instruction is contained within a basic block (see Figure 2). Since the execution of one operation of a basic block implies the execution of the whole block (under most circumstances) all operations in a block are equally suspicious under any coverage-based statistical suspiciousness measure. Thus, it suffices to compute the suspiciousness for a block as a whole rather than doing so separately for each statement or instruction in the block.

One method of measuring program behavior is through flow graph profiling (e.g., as performed by Dynagrok). In this paper a "suspicious behavior" is a subgraph h of a dynamic control flow graph (DCFG) g such that execution of h is statistically associated with program failure. The framework outlined above will be extended from particular basic blocks to subgraphs of flow graphs. This allows nearly any CBSFL suspiciousness measure to be re-used as a suspiciousness measure for flow graph fragments.

TABLE I: Probability estimators adapted to subgraphs of basic block flow graphs.

Probability Estimator	Formula
$\widehat{\Pr}[F]$	$\frac{f}{n}$
$\widehat{\Pr}[P]$	$\frac{p}{n}$
$\widehat{\Pr}[F \cap h]$	$\frac{\lvert\{g : g \in \mathcal{F} \wedge h \sqsubseteq g\}\rvert}{n}$
$\widetilde{\Pr}[P \cap h]$	$\frac{\sum_{\epsilon \in E_h} \widehat{\Pr}[P \cap \epsilon] + \sum_{v \in V_h} \widehat{\Pr}[P \cap v]}{\lvert E_h\rvert + \lvert V_h\rvert}$
$\widehat{\Pr}[h]$	$\widehat{\Pr}[F \cap h] + \widetilde{\Pr}[P \cap h]$

TABLE II: A representative set of suspiciousness metrics for statistical fault localization [24] defined in terms of probability estimators [26].

Suspiciousness Metric	Formula
Precision	$\frac{\Pr[F \cap h]}{\Pr[h]}$
F1	$2\frac{\Pr[h]}{\Pr[F] + \Pr[h]} \frac{\Pr[F \cap h]}{\Pr[h]}$
Ochiai	$\sqrt{\frac{\Pr[F \cap h]}{\Pr[F]} \frac{\Pr[F \cap h]}{\Pr[h]}}$
Jaccard	$\frac{\Pr[F \cap h]}{\Pr[F] + \Pr[P \cap h]}$
Information Gain	$\left[\frac{\Pr[F \cap h]}{\Pr[h]} \log_2\left(\frac{\Pr[F \cap h]}{\Pr[h]}\right) + \frac{\Pr[P \cap h]}{\Pr[h]} \log_2\left(\frac{\Pr[P \cap h]}{\Pr[h]}\right)\right] - [\Pr[F] \log_2(\Pr[F]) + \Pr[P] \log_2(\Pr[P])]$
Associational Risk	$\frac{\Pr[F \cap h] - \Pr[F]\Pr[h]}{\epsilon + \Pr[h] - (\Pr[h])^2}$
Contrast	$\Pr[F \cap h] - \Pr[P \cap h]$
Relative-Precision	$\frac{\Pr[F \cap h]}{\Pr[h]} - \Pr[F]$
Relative-F1	$2\frac{\Pr[h]}{\Pr[F] + \Pr[h]} \left(\frac{\Pr[F \cap h]}{\Pr[h]} - \Pr[F]\right)$
Relative-Ochiai	$\sqrt{\frac{\Pr[h]}{\Pr[F]}} \left(\frac{\Pr[F \cap h]}{\Pr[h]} - \Pr[F]\right)$
Relative-Jaccard	$\frac{\Pr[F \cap h]}{\Pr[F] + \Pr[P \cap h]} - \Pr[F]$

As before, the probability of program failure is $\Pr[F] \approx \frac{f}{n}$ and the probability of a program not failing is $\Pr[P] \approx \frac{p}{n}$. However, our other two "building block" estimators will need to be modified for use with subgraphs. Let $\mathcal{F}$ be the set of DCFGs collected from failing executions and let $\mathcal{P}$ be the

set from non-failing (passing) executions. Let g be a dynamic control flow graph, and let h be a subgraph of g, denoted $h \sqsubseteq g$. We say that the subgraph h is "covered" by any program execution with DCFG g.

Equation 2 defined the Ochiai metric in terms of the probabilities $\Pr[F \cap l]$ (the probability that the program fails and the location l is executed) and $\Pr[P \cap l]$ (the probability that the program does not fail and the location l is executed). Ochiai can be adapted for use with subgraphs by replacing these probabilities with analogous ones: $\Pr[F \cap h]$ (the probability that the program fails and subgraph h is covered) and $\Pr[P \cap h]$ (the probability that the program does not fail and subgraph h is covered). Estimators for these probabilities can be defined as:

$$\Pr[F \cap h] \approx \frac{|\{g : g \in \mathcal{F} \wedge h \sqsubseteq g\}|}{n} \quad (3)$$

$$\Pr[P \cap h] \approx \frac{|\{g : g \in \mathcal{P} \wedge h \sqsubseteq g\}|}{n} \quad (4)$$

Ochiai can then be redefined for suspicious subgraphs (behaviors) as:

$$Och_{\mathcal{F},\mathcal{P}}(h) \approx \sqrt{\frac{\Pr[F \cap h]}{\Pr[F \cap h] + \Pr[P \cap h]} \times \frac{\Pr[F \cap h]}{\Pr[F]}} \quad (5)$$

The other suspiciousness measures discussed in a recent paper by Sun and Podgurski [26] can be adapted in a similar fashion (Table II adapts a representative set of the better performing metrics).

Many suspicious behaviors never appear in full among the dynamic flow graphs of the non-failing (passing) executions. However, portions of these behaviors do appear. The estimator above for $\Pr[P \cap h]$ will always estimate the probability of such subgraphs as 0. This seems to be an underestimate for subgraphs for which a majority of their vertices and edges are covered by non-failing executions. An alternative (and efficiently computable) estimator $\widetilde{\Pr}[P \cap h]$ averages the probability estimates for each edge and vertex:

$$\widehat{\Pr}[P \cap \epsilon] = \frac{|\{g : g \in \mathcal{P} \wedge \epsilon \in E_g\}|}{n}$$

$$\widehat{\Pr}[P \cap v] = \frac{|\{g : g \in \mathcal{P} \wedge v \in V_g\}|}{n}$$

$$\widetilde{\Pr}[P \cap h] = \frac{\sum_{\epsilon \in E_h} \widehat{\Pr}[P \cap \epsilon] + \sum_{v \in V_h} \widehat{\Pr}[P \cap v]}{|E_h| + |V_h|} \quad (6)$$

This new estimator gives "partial credit" to a graph h which has substructures which are covered by passing executions.

Table I summarizes the definitions of the probability estimators used throughout the rest of this chapter. Table II provides the formulas for a representative set of suspiciousness metrics adapted for use with DCFG subgraphs.

IV. Background: Mining Suspicious Behaviors

The goal of discriminative or suspicious pattern mining is to extract from "positive" and a "negative" datasets patterns which distinguish between the two sets. In Suspicious-Behavior-Based Fault Localization (SBBFL) the datasets are sets of Dynamic Control Flow Graphs (DCFGs) collected from test executions. One set $\mathcal{F}$ contains the DCFGs collected from executions that failed. The second set $\mathcal{P}$ contains the DCFGs from passing (or non-failing) executions of the program.

Because the datasets $\mathcal{F}$ and $\mathcal{P}$ are sets of graphs, the distinguishing patterns will be graphs. In SBBFL, the identified patterns are all subgraphs of the graphs collected from failing executions. The extracted patterns are statistically correlated with program failure, as in Coverage-Based Statistical Fault Localization (CBSFL). Some of the options for measuring the correlation between program failure and the presence of a subgraph in a DCFG were discussed in Section III.

Finding these suspicious subgraphs, is an application of *discriminative (or significant) subgraph mining* as solved by the Branch-And-Bound (B&B) family of algorithms [27]–[30]. In significant subgraph mining the goal is to find the most significant subgraph(s) according to some measure of significance [31]. If there are multiple classes of graphs in the database (e.g. "positive" and "negative" graphs), significance measures such as Information Gain [7], [30] are used to guide the algorithm to find subgraphs that *discriminate* between $\mathcal{F}$ and $\mathcal{P}$.

Previous SBBFL studies [7], [13] used algorithms in the B&B family [27] (such as LEAP Search [30]) to mine the top-k suspicious subgraphs.

Definition 1 (Top-k Suspicious Subgraph Mining)**.** ***Given*** *an integer $k > 0$; $\mathcal{F}$, the set of DCFGs collected from failing program executions; $\mathcal{P}$, the set of DCFGs from passing executions; and a suspiciousness measure ς:* ***find*** *a set of subgraphs H such that $|H| = k$ and $\sum_{h \in H} \varsigma(h)$ is maximized.*

In principle the suspiciousness measure ς in the definition above could use arbitrary information from a subgraph h and the sets $\mathcal{F}$ and $\mathcal{P}$. However, in symmetry with the discussion in Section III and with previous work [30] we will only consider measures defined in terms of the probability estimators in Table I. Since for any given sets $\mathcal{F}$ and $\mathcal{P}$ the estimators $\widehat{\Pr}[F]$ and $\widehat{\Pr}[P]$ are fixed, ς can be viewed as a function of $\widehat{\Pr}[F \cap h]$ and $\widetilde{\Pr}[P \cap h]$: $\varsigma(\widehat{\Pr}[F \cap h], \widetilde{\Pr}[P \cap h]) \rightarrow \Re$.

The B&B algorithms solve the Top-k Suspicious Subgraph Mining problem by enumerating subgraphs of graphs in the set $\mathcal{F}$. To prune portions of the search space they compute a mathematical upper bound $\hat{\varsigma}$ on the suspiciousness measure ς. Here, we will focus on the computation of the upper bound on the suspiciousness (or significance) score for all potential supergraphs of any particular subgraph h. (A more detailed description of the algorithm is given in Section VII.) One such bound for ς (which improves upon the upper bound in [30]) can be constructed using the following two facts:

1) A user may specify a subgraph h appears among the graphs in $\mathcal{F}$ at least f_{min} times. Then $\frac{1}{f_{\text{min}}}$ becomes a lower bound on $\Pr[F \cap h]$.
2) The user may specify a maximum number of edges $|E|_{\text{max}}$

allowed in a subgraph, giving a lower bound on $\widetilde{\Pr}[P \cap h]$:

$$\widetilde{\Pr}[P \cap h]_{\min} = \frac{\sum_{\epsilon \in E_H} \widehat{\Pr}[P \cap \epsilon] + \sum_{v \in V_H} \widehat{\Pr}[P \cap v]}{2|E|_{\max} + 1}$$

These facts can be used to derive a lower bound on $\widetilde{\Pr}[P \cap h]$. Combining the two lower bounds yields an upper bound $\hat{\varsigma}$ on ς for the supergraphs of h:

$$\hat{\varsigma} = \max \left\{ \begin{array}{l} \varsigma(\Pr[F \cap h], \widetilde{\Pr}[P \cap h]_{\min}), \\ \varsigma(\frac{1}{f_{\min}}, \widetilde{\Pr}[P \cap h]) \end{array} \right\} \quad (7)$$

In order for Equation 7 to be used as bound within a B&B algorithm, ς must satisfy a technical property (Equation 5 in the LEAP Search paper [30]) that we will call the *Discriminative Velocity Property* (DVP). For a metric ς to satisfy DVP its partial derivatives with respect to $\Pr[F \cap h]$ and $\Pr[P \cap h]$ must meet certain requirements. This ensures the metric ς is increasing in the correct circumstances. Due to space limitations and the complexity of the property we will not go into the mathematical details – see the LEAP Search paper [30].

Table III shows that, of the metrics in Table II, only Information Gain satisfies DVP. Because satisfaction of DVP is a requirement for using a metric with a B&B algorithm, of the metrics shown in Table II only Information Gain can be used in a B&B algorithm. As we will see in the Empirical Evaluation in Section VI, this is a major limitation of the B&B algorithms because Information Gain turns out to be one of three metrics which perform markedly worse than the other metrics considered. The next section develops a solution to this problem in the form of a new type of algorithm which does not rely on an explicit enumeration of the search space.

V. Sampling Suspicious Behaviors

Branch-And-Bound (B&B) algorithms have four drawbacks when applied to automatic fault localization:

1) Suspiciousness metrics must satisfy DVP, restricting B&B to metrics such as Information Gain (see Table III).
2) B&B algorithms enumerate the subgraphs of graphs in $\mathcal{F}$ — a scalability bottleneck [31].
3) The user must specify the maximum number of edges allowed in a subgraph (from Eq. 7).
4) The user must specify the maximum number of subgraphs to mine (from Def. 1).

In order to solve the aforementioned problems with the Branch-And-Bound framework, we have developed a new algorithm, called *Score Weighted Random Walks* (SWRW), for finding suspicious, significant, or discriminative subgraphs. The new method approximates Branch-And-Bound's output by sampling the search space. The sampling is weighted by the suspiciousness scores, which (heuristically) minimizes the error from Branch-And-Bound's output. A sampling approach makes it easy to trade-off computation time with accuracy by adjusting the sample size.

The new method, unlike the B&B algorithms, does not require that suspiciousness metrics satisfy DVP — which most metrics *do not satisfy*. Instead, metrics must satisfy a new, less restrictive property we call the *Inverse Velocity Property* (IVP). As shown in Table III, all but two of the suspiciousness metrics shown in Table II satisfy IVP.

Definition 2 (Inverse Velocity Property)**.** *Let ς be a suspiciousness score, $x = Pr[F \cap h]$, and $y = Pr[P \cap h]$. The Inverse Velocity Property is satisfied if:*

$$(x \in (0,1] \wedge y \in [0,1]) \implies \left(\frac{\partial \varsigma}{\partial x} > 0 \wedge \frac{\partial \varsigma}{\partial y} < 0 \right)$$

IVP requires a suspiciousness metric to increase as the support of a subgraph h either increases in the set $\mathcal{F}$ of graphs from failing executions or decreases in the set $\mathcal{P}$ of graphs from passing executions. It also requires the metric to decrease when support for h falls in $\mathcal{F}$ or increases in $\mathcal{P}$. This follows the anti-monotonic structure of subgraph mining [31]:

$$h \sqsubseteq h' \implies \varsigma(h) \leq \varsigma(h')$$

If SWRW is run using a metric that does not satisfy IVP, it will return results that are incorrect – either missing graphs it should have found or including graphs it should not have found. Like DVP for Branch-And-Bound, IVP is *required* for the output of SWRW to be correct.

Suspiciousness metrics which satisfy IVP induce a (newly defined) *suspicious subgraph lattice* on the subgraphs of the graphs in $\mathcal{F}$. Figure 3 shows an example suspicious subgraph lattice for a small undirected graph dataset. SWRW samples suspicious subgraphs from the lattice via a weighted forward random walk. The suspicious subgraph lattice is a graph whose nodes represent subgraphs of the dataset. It is a subgraph of the *connected subgraph lattice* [32].

Definition 3 (Connected Subgraph Lattice of $\mathcal{G}$)**.** *The subgraph relation $\cdot \sqsubseteq \cdot$ induces the* Connected Subgraph Lattice *$\mathcal{L}_{\mathcal{G}}$ representing all the possible ways of constructing a graph $G \in \mathcal{G}$ from the empty subgraph by adding one edge at a time. $\mathcal{L}_{\mathcal{G}}$ is a digraph where each vertex u represents a unique connected (ignoring edge direction) subgraph of some $G \in \mathcal{G}$. There is an edge from u to v in $\mathcal{L}_{\mathcal{G}}$ if adding some edge ϵ to u creates a subgraph $u + \epsilon$ isomorphic to v, $v \cong u + \epsilon$.*

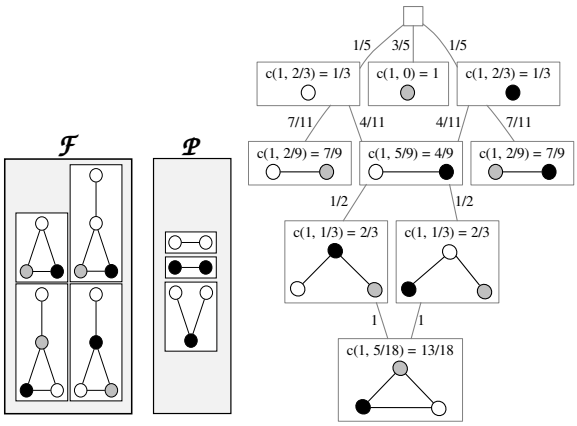
Definition 4 (Suspicious Subgraph Lattice)**.** *The subgraph relation $\cdot \sqsubseteq \cdot$ and a suspiciousness measure ς satisfying IVP induce a* Suspicious Subgraph Lattice *ς-$\mathcal{L}_G$. The lattice ς-$\mathcal{L}_G$ is a connected subgraph of the connected subgraph lattice $\mathcal{L}_{\mathcal{G}}$ rooted at the root of $\mathcal{L}_{\mathcal{G}}$. Let the empty subgraph $h_\varnothing$ be in $V_{\varsigma\text{-}\mathcal{L}_G}$ as the root node of ς-$\mathcal{L}_G$ (it is also the root of $\mathcal{L}_{\mathcal{G}}$). If an edge $(u, v) \in E_{\varsigma\text{-}\mathcal{L}_G}$ then v is in $V_{\varsigma\text{-}\mathcal{L}_G}$. An edge $(u, v) \in E_{\mathcal{L}_{\mathcal{G}}}$ is an edge in $E_{\varsigma\text{-}\mathcal{L}_G}$ if and only if:*

$$u \in V_{\varsigma\text{-}\mathcal{L}_G} \wedge \left[\varsigma(u) < \varsigma(v) \vee \varsigma(u) = \varsigma(v) \wedge \frac{Pr[F \cap v]}{Pr[v]} = 1 \right]$$

Listings 1 and 2 show the new algorithm, Score Weighted Random Walk, for finding suspicious behaviors by sampling maximal suspicious subgraphs from the suspicious subgraph lattice ς-$\mathcal{L}_G$. The algorithm is built around a core function (`swrw` in Listing 1). The algorithm takes a random walk on an

absorbing Markov chain [33], [34] built from the suspicious subgraph lattice ς-$\mathcal{L}_G$. A Markov chain is *absorbing* if an absorbing state is reachable from every state, where a state is absorbing if it cannot be left once it is entered. The random walk taken by SWRW is weighted by the suspiciousness scores ς of the subgraphs in the lattice, which causes it to visit the more suspicious subgraphs more frequently. When the walk reaches an absorbing state the graph represented by the state is sampled.

The function `swrw` in Listing 1 simulates a Markov process starting at some state in the chain (`start`) and transitioning until it is absorbed at an absorbing state. The absorbing state, which is a suspicious subgraph, is then returned as the sample. At each step in the simulation there are four sub-steps. The first is to check for a random restart of the walk. The second sub-step computes the supergraphs of the graph `cur` that represents the current state of the Markov chain. The next sub-step filters



(a) Dataset of graphs (b) Suspicious subgraph lattice

Fig. 3: Example suspicious subgraph lattice constructed using the Contrast suspiciousness metric (see Table II). The colors stand in for labels in this simple example. In each lattice node (the boxes) the suspiciousness scores are shown ("c" stands for Contrast). On the lattice edges (edges between the boxes) the *forward* transition probabilities are shown.

TABLE III: DISCRIMINATIVE VELOCITY PROPERTY (DVP) (EQ. 5 IN [30]) AND INVERSE VELOCITY PROPERTY (IVP) (DEF. 2) SATISIFACTION FOR EACH SUSPCIOUSNESS METRIC IN TABLE II.

Suspiciousness Metric	Satisfies DVP	Satisfies IVP
Precision		✓
F1		✓
Ochiai		✓
Jaccard		✓
Information Gain	✓	✓
Associational Risk		
Contrast		✓
Relative-Precision		✓
Relative-F1		✓
Relative-Ochiai		
Relative-Jaccard		✓

Note: Satisfaction was checked using Mathematica for most measures but had to be checked by hand for Relative-Ochiai and Information Gain. Information Gain was previously shown to satisfy DVP [7], [30] and our analysis confirmed this result. None of the other metrics satisfy DVP.

the supergraphs such that only those in $V_{\varsigma\text{-}\mathcal{L}_G}$ are kept. Finally, the function `weighted_sample` is used to select the next state for the Markov process to transition to.

To collect multiple samples from the suspicious subgraph lattice, k random walks could be taken. This approach is shown in function `k_walks` in Listing 2. Each walk starts from the bottom of the lattice and, using the supplied walk function (e.g.

```
# param F: The flow graphs collected from failing executions
# param P: The flow graphs collected from passing executions
# param score: The suspicousness measure
# param min_F_sup: The minimum number of graphs in the set F
#                  that a suspicious graph must appear in
# param start: the graph in the suspicious subgraph lattice
#              to start the walk from.
# returns a subgraph of F randomly sampled from the
# suspicious subgraph lattice induced by P and score.
def swrw(F, P, score, min_F_sup, start):
    cur = start
    prev = cur
    while cur is not None:
        # At each step in the walk, a random restart occurs
        # with probability proportional to the maximum length
        # of the walk.
        if random.random() < 1.0/MAX_EDGES:
            cur = start; prev = cur
            continue
        # Compute the direct supergraphs of the current
        # graph
        supers = extend_with_one_edge(F, P, cur)
        # Filter out graphs not in the suspicious subgraph
        # lattice
        supers = filter_supergraphs(F, P, min_F_sup, score,
                cur, supers)
        # Randomly select a supergraph to be the next graph
        # in the walk, favoring those with higher
        # suspiciousness.
        prev = cur
        cur = weighted_sample(score, supers)
    return prev

# Filters out graphs in supers that are not in the
# suspicious subgraph lattice
# param h: the current subgraph (which is in the SSL)
# param supers: supergraphs of h
# returns a subset of supers
def filter_supergraphs(F, P, min_F_sup, score, h, supers):
    allowed = list()
    for sg in supers:
        precision = Pr(F, sg)/(Pr(F, sg) + Pr(P, sg))
        if score(sg) == score(h) and precision == 1:
            allowed.append(sg)
        elif score(sg) > score(h):
            allowed.append(sg)
    return allowed

# Using the score to weight the graphs sample one graph
# from graphs
# param score: the weighting score
# param graphs: a list of graphs
# returns one graph from graphs or None if graphs was empty
def weighted_sample(score, graphs):
    if len(graphs) <= 0:
        return None
    if len(graphs) == 1:
        return graphs[0]
    min_weight = min(score(g) for g in graphs)
    shift = max(0, -min_weight)
    weights = [ score(g) + shift + 1e-8 ## ensure > 0
                for g in graphs ]
    total = sum(weights)
    i = 0
    r = total * random.random()
    while i < len(weights) - 1 and r > weights[i]:
        r -= weights[i]
        i += 1
    return i
```

Listing 1: Python psuedocode for the new SWRW algorithm.

swrw), samples a suspicious subgraph from the lattice. This approach begs the question of how to choose an appropriate value for k.

The function walk_top_vertices in Listing 2 provides an answer. Instead of taking k walks starting from the root node, it takes walks starting from a percentage p of subgraphs representing single vertices — suspicious locations — in the graphs in $\mathcal{F}$. The function orders the subgraphs in decreasing order from from most suspicious to least suspicious. This ensures, by the Inverse Velocity Property, that SWRW starts from vertices of ς-$\mathcal{L}_G$ that are likely to lead to the most suspicious subgraphs.

VI. Empirical Evaluation

We empirically evaluated SWRW's fault localization performance in terms of accuracy and cost. SWRW was compared to a very good previously proposed behavioral fault localization approach [7] that used LEAP Search [30] – the fastest known significant subgraph mining algorithm [31]. The following questions were considered:

1) Which suspiciousness metric (of those in Table II) provides the most accurate fault localization? Do the alternative suspiciousness metrics provide better fault localization performance than Information Gain? Variants of Information Gain predominate in past work on behavioral fault localization [7], [9], [10], [12], [13].
2) Comparing SWRW to Branch-And-Bound and sLeap (both of which require the Information Gain metric), which algorithm provides the best fault localization performance?

```
# param F: The flow graphs collected from failing executions
# param P: The flow graphs collected from passing executions
# param score: The suspicousness measure
# param min_F_sup: The minimum number of graphs in the set F
#                  that a suspicious graph must appear in
# param k: the number of walks to take
# returns a set of at most k subgraphs
def k_walks(F, P, score, min_F_sup, k):
    return {
        swrw(F, P, score, min_F_sup, F.empty_subgraph())
        for _ in xrange(k)
    }

# param F: The flow graphs collected from failing executions
# param P: The flow graphs collected from passing executions
# param score: The suspicousness measure
# param min_F_sup: The minimum number of graphs in the set F
#                  that a suspicious graph must appear in
# param p: the percentage of the top scoring vertices to
#          start walks from
# param w: the number of walks to take from each vertex
# returns a set of suspicious subgraphs
def walk_top_vertices(F, P, score, min_F_sup, p, w):
    # Sort the unique vertices in F by their scores.
    vertices = F.unique_vertices()
    vertices.sort(key=lambda v: score(v))
    # Take walks starting from a percentage p of the
    # vertices, starting with the most suspicious vertices.
    subgraphs = set()
    for i, v in enumerate(vertices):
        if i >= p * len(vertices):
            break
        # Take w walks from each vertex
        for _ in xrange(w):
            subgraphs.add(swrw(F, P, score, min_F_sup, v))
    return subgraphs
```

Listing 2: Python psuedocode for collecting multiple subgraphs using SWRW.

A. Methodology

A new dataset of five real world programs with injected mutation faults was created. The programs are all written in the Go programming language. Dynagrok (see Section II) was used to inject the mutations and instrumentation into them. The injected mutations were all branch condition mutations which flip the condition (ex. `if true` → `if false`). Such mutations are sufficient for the purposes of this study, because the resulting faults are favorable to localization by coverage based fault localization techniques in general, including all of the techniques considered in this paper. The mutations created do not favor any techniques being compared over any of the other techniques.

While the bugs used in the study were exclusively randomly inserted mutation faults the *tests* were real test cases. For the 4 large programs we used either real world operational inputs sampled from the internet (for the HTML parser and the Markdown processor) or from the included integration tests provided by the library developers (for the Go compiler and the Javascript interpreter). For the last and smallest program (an AVL tree) we used randomly generated tests that were sequences of AVL tree operations (Put, Has, Get, and Remove). Table IV summarizes the subject programs used in the study.

To assess the performance of Suspicious-Behavior Based Fault Localization, we used a fault rank cost measure similar to those used in previous SBBFL studies [7], [8], [13] and analogous to a measure used in CBSFL studies [26]. The behaviors (subgraphs) are scored using a suspiciousness metric and presented to the programmer in ranked order with the most suspicious subgraph first. The fault rank gives the expected number of behaviors a programmer would examine before examining a behavior containing the faulty location(s). The fault rank gives an objective score enabling comparison between different suspiciousness metrics for the same program version and between different programs and versions. Note: it is not appropriate to directly compare standard CBSFL techniques to SBBFL techniques using this cost measure.

All of the algorithms compared in the study have an element of randomness to them. For instance, Branch-And-Bound algorithms make random choices about which subgraph to keep when some have equal scores. SWRW is a sampling algorithm and is explicitly randomized. Thus, all experiments were replicated and their results were averaged. Sometimes the algorithms return no behaviors that contain a faulty location — a localization failure. To incorporate localization failures into

TABLE IV: Datasets used in the evaluation

Program	L.O.C.	Mutants	Description
AVL (github.com/timtadh/dynagrok)	483	19	An AVL tree
Blackfriday (github.com/russross/blackfriday)	8,887	19	Markdown processor
HTML (golang.org/x/net/html)	9,540	20	An HTML parser
Otto (github.com/robertkrimen/otto)	39,426	20	Javascript interpreter
gc (go.googlesource.com/go)	51,873	16	The Go compiler

Note: The AVL tree is in the examples directory.

the overall average performance, the maximum fault rank for the relevant program (across all algorithms) is used as the fault rank when none of the returned behaviors contain the fault. Using the maximum fault rank as the cost of a localization failure means that when evaluating the performance of the Branch-And-Bound algorithms the average fault rank may be higher than the number of subgraphs mined. Under the debugging model assumed by these cost measures, programmers check each behavior (or location for CBSFL) before moving on to the next one in the list. Thus, a programmer using a SBBFL tool that fails to localize a bug would need to examine all of the mined graphs and then continue with traditional debugging methods.

B. Limitations and Threats to Validity

This study only considered suspicious-behavior based fault localization (SBBFL) algorithms. *The results are intended to indicate which of the algorithms considered performed the best at the fault localization task, not whether SBBFL outperforms CBSFL or other fault localization methods*. Further work is needed to fully evaluate SBBFL algorithms against other approaches to fault localization (statistical or otherwise). While our study tried to consider a representative set of suspiciousness metrics, many other metrics are available [24] and the results shown here may not fully generalize to those metrics. Similarly, although the programs used in the study were chosen to represent programs with varying degrees of complexity, it was also necessary to choose programs with readily available inputs, which limits the generalizability of the results.

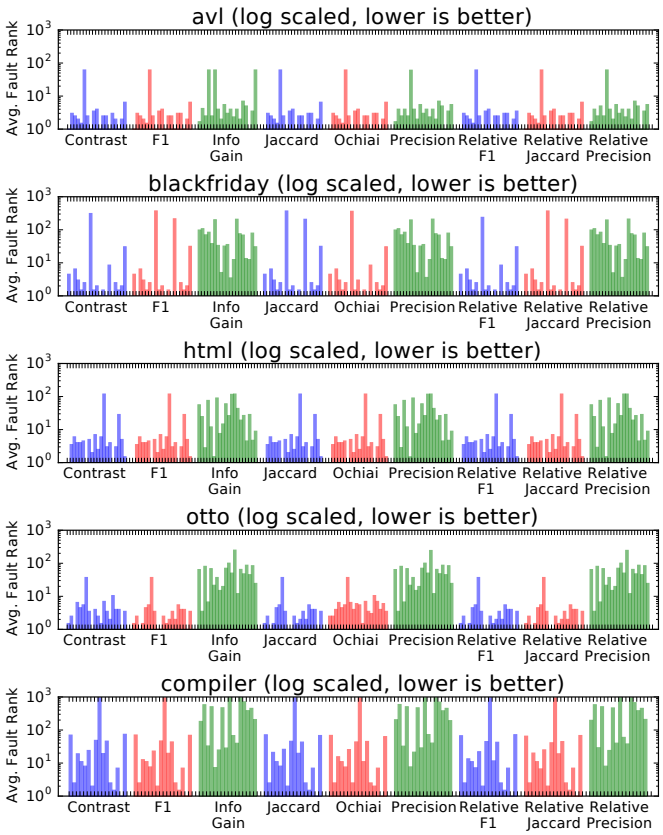


Fig. 4: Fault localization performance of SWRW (log scaled, lower is better). Each bar gives the average fault rank for a particular buggy program version and suspiciousness metric from Table II. SWRW was configured to use `walk_top_vertices` from Listing 2 with $p = .2$ and $w = 2$. Subgraphs with at most 100 edges were collected. The experiment was repeated 20 times and the mean results are shown.

TABLE V: THE MEAN FAULT RANK FOR EACH METRIC IN FIG. 4.

Ochiai	40.6	Contrast	40.1	Precision	102.1
Jaccard	42.6	Relative Jaccard	42.5	Relative Precision	101.8
F1	42.6	Relative F1	**38.8**	Information Gain	131.3

The gold standard for evaluating software testing techniques is to use real bugs with real test cases during the evaluation [35], [36]. In this paper, we are introducing a new language (Go) and analysis platform (Dynagrok) to the software engineering community, in addition to demonstrating a new algorithm (SWRW) for fault localization. For long-established programming languages such as Java there are many projects available with bug databases that are suitable for use in fault localization studies – such as Defects4J [37]. In the future, the authors believe such datasets should be constructed for Go as well. For Go, this paper makes progress on tooling, in the expectation that the datasets will come. In the meantime, our new technique, SWRW, is evaluated using *real test cases* on bugs which were injected via program mutation.

Another important threat to validity (for both our study and previous studies [7], [8], [13]) is the use of the fault rank cost measure. Recall that this cost measure is the expected number of behaviors a programmer would consider before considering a behavior containing the faulty location(s). This measure is simple to compute and explain but it does not consider the variable effort a programmer might need to expend on behaviors of different sizes. Next, while the expectation takes into account the faulty location appearing in multiple behaviors it doesn't take into account a programmer skipping behaviors that contain locations the programmer has already examined. We utilize this metric as way to compare the SBBFL performance between similar algorithms, not to demonstrate the overall superiority of SBBFL or SWRW in particular.

C. Which Suspiciousness Measure Works the Best?

Figure 4 shows the fault localization performance of SWRW for all applicable suspiciousness metrics across all program versions. Table V gives the average ranks for each metric across all programs. SWRW was chosen because the other algorithms (B&B and sLeap) only support Information Gain. For all programs except the AVL tree, Information Gain, Precision, and Relative Precision all performed markedly worse than the other metrics. Contrast, F1, Jaccard, Ochiai, Relative F1, and Relative Jaccard all performed about the same (with some small differences). This answers RQ1: the best metrics to use are Contrast, F1, Jaccard, Ochiai, Relative F1, or Relative Jaccard. Information Gain, despite being the metric of choice in previous studies, was not competitive with five of the other suspiciousness metrics considered in this empirical study. As

shown in Table V, Information Gain had the worst average performance of all of the metrics considered.

D. Which Algorithm Performs the Best?

Figure 5 compares Branch-And-Bound (B&B), sLeap, and SWRW in various configurations using the Information Gain metric — the only metric under consideration that B&B and sLeap can use. B&B and sLeap both have two versions. The first version is the one discussed in Section IV. The second version (denoted "B&B (max)" and "sLeap (max)") imitates the SWRW's behavior during their search by having them (only) traverse the suspicious subgraph lattice. SWRW was run in 4 configurations: `k-walks 100` (demonstrating `k_walks` with $k = 100$ from Listing 2) and 3 configurations of `walk-top-vertices` (demonstrating `walk_top_vertices` from Listing 2 with: $p = .2$, $w = 2$; $p = 1$, $w = 2$; and $p = 1$, $w = 10$).

The second graph from the top in Figure 5 shows the average fault localization performance for each algorithm on three of the datasets: avl, blackfriday and otto. The third and fourth graphs detail the fault localization performance

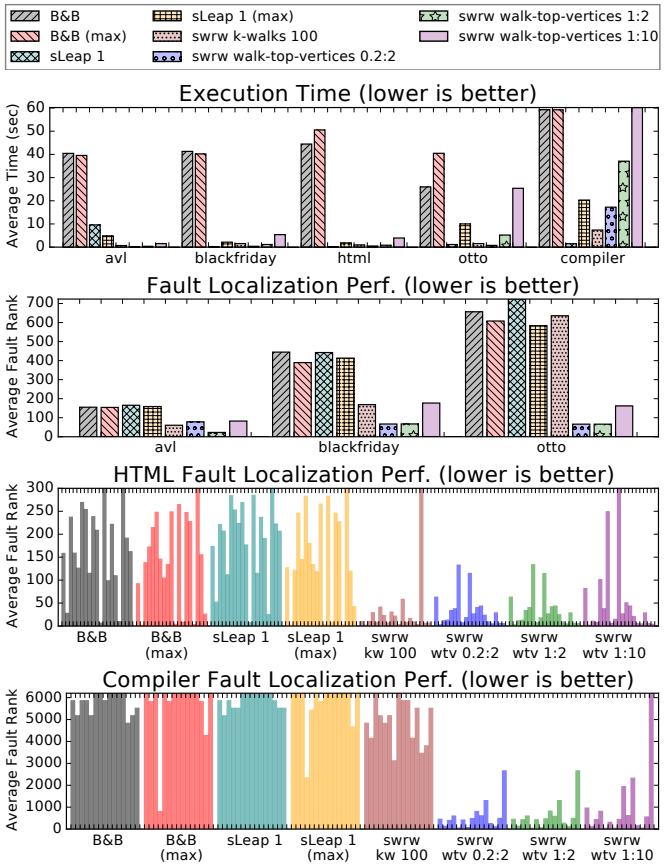


Fig. 5: Comparision of Branch-And-Bound (B&B), sLeap, and SWRW using Information Gain. In the first and second graphs each bar represents the average performance for all program versions. The third and fourth graphs "zoom in" on the fault localization performance for the HTML Parser and the Go Compiler. Branch-And-Bound and sLeap collected 50 subgraphs each. All algorithms collected subgraphs with at most 15 edges. A 60 second timeout was set for all executions. The experiment was repeated 20 times and the mean results are shown. Note: "kw" = k-walks, "wtv" = walk-top-vertices.

for HTML and the Go compiler respectively. SWRW in all configurations performed significantly better than B&B and sLeap. B&B and sLeap mined 50 subgraphs and often failed to mine any subgraph containing the fault. Increasing k to allow B&B and sLeap to mine more subgraphs did not help as they would not reliably terminate. In comparison, SWRW usually sampled a subgraph containing the fault. This answers RQ2: SWRW provided the best fault localization performance using Information Gain. As noted above, it provided even better performance when alternative suspiciousness metrics are used.

The top graph in Figure 5 shows the average execution time for each algorithm on each dataset. The default configuration of SWRW (corresponding to the blue bar with circles) was able to extract the suspicious behaviors in less than 2 seconds for all of the datasets except that for the Go Compiler (for which it took 20 seconds). In comparison, B&B often timed out and had an average execution time of 30 seconds or more. sLeap performed much better than B&B (thanks to its pruning heuristic) and was competitive with SWRW. However, as shown in the first, third, and fourth graphs, SWRW had much better fault localization performance.

E. Summary of Results

SWRW outperformed the other discriminative subgraph mining algorithms at behavioral fault localization. Information Gain, Precision, and Relative Precision all performed markedly worse than the other suspiciousness metrics. The previous mining algorithms use the Information Gain metric, and a technical restriction (see Section IV) prevents them from using any of the other metrics considered in Table II. Since SWRW is not bound by the same restrictions as algorithms in the Branch-And-Bound family, SWRW appears to be a better and more flexible algorithm to use for behavioral fault localization.

VII. Related Work

A. Fault Localization

There have been a number of studies [3], [7]–[15], [19], [38] combining the statistical fault localization approach with graph mining. We will discuss a few of representative studies. C. Liu, X. Yan, H. Yu, J. Han, and P. Yu [3] first introduced the idea of mining program behaviors to localize faults. Xifeng Yan and Jiawei Han had previously created the classic frequent subgraph mining algorithms gSpan [39] and CloseGraph [40] but had yet to create LEAP Search [30]. The study used CloseGraph to extract frequent subgraphs of dynamic call graphs, which were then used as features in a classifier. The classifier was then used to identify the functions in the program which were most relevant to the fault.

Di Fatta *et al.* [8] uses a frequent subtree mining algorithm (FREQT) to extract frequently occurring subtrees from the dynamic call trees. They then used the Precision metric (see Table II) to score each frequent subtree. Di Fatta *et al.* encountered scalability limitations when mining call trees which they solved by tightly limiting the size of the extracted subtrees.

Eichinger conducted multiple studies [9], [10], [12] using CloseGraph [40] as implemented by Wörlein [41] in ParSeMiS. In Eichinger's studies, dynamic call trees are collected and then reduced into weighted dynamic call graphs. The frequent subgraphs are extracted using CloseGraph and then scored using variants of Information Gain. The Information Gain variants make use of the call weights [9] and discretized function parameter values [10]. Eichinger employed a hierarchical [12] approach for multi-level (package, class, and method) localization. A similar multi-level behavioral localizer is HOLMES [19] which also employs an adaptive profiler. However, instead of collecting call trees or call graphs HOLMES collects path profiles [42].

H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan [7] used LEAP Search [30] by X. Yan to localize faults. The faults were localized to suspicious behaviors which are subgraphs of either dynamic control flow graphs or dynamic call graphs. Cheng extended the original LEAP Search algorithm to collect the Top-k subgraphs rather than just 1 subgraph. Their extended algorithm is covered in detail in Section IV. Cheng compares the results of their approach to RAPID [43] which uses frequent sub-sequence mining (a cousin of frequent subgraph mining) to extract frequently recurring sub-traces from program execution traces.

Parsa *et al.* [13] presented an alternative algorithm to LEAP Search [30] for finding the top-k discriminative subgraphs for fault localization. Their algorithm fits into the Branch-And-Bound framework and they claim similar performance to LEAP Search. They define their own suspiciousness metric which they call the $\mathcal{F}_{\text{Score}}$. Since their algorithm is a Branch-And-Bound algorithm they also provide an upper bound for their score.

B. Graph Mining

Significant subgraph mining [27]–[30] is an alternative to the frequent subgraph mining problem [31]. Significant subgraph mining finds the most important subgraphs as judged an objective function. Kudo *et al.* presented gBoost [27] which integrates discriminative subgraph mining into gSpan [39] — a classic frequent subgraph mining algorithm — introducing the first Branch-And-Bound algorithm. Saigo *et al.* [28] applied partial least squares regression to graph data with gPLS. gPLS uses a modified Branch-And-Bound algorithm to extract the patterns which are used as features in a regression. Finally, Thoma *et al.* [29] presented CORK which integrates a different pruning operator into gSpan than gBoost used. CORK runs gSpan in a loop. Each step of the loop it expands the set of discriminative graphs by one graph until no improvement is made.

The Branch-And-Bound (B&B) family of algorithms [27]–[30] enumerates the subgraphs of $\mathcal{F}$ in a depth-first manner. At each step of the algorithm Branch-And-Bound considers a subgraph h of a graph $g \in \mathcal{F}$. Using a suspiciousness/importance measure (ς above) B&B scores h. If h's score higher than best score found so far h becomes the exemplar $\bar{h}$. B&B algorithms then compute supergraphs h' of h (where $h' \sqsubseteq g$ for a graph $g \in \mathcal{F}$). For each h', B&B checks to see if *any supergraph* of h' could have a score at least as large as the exemplar $\bar{h}$ by computing an upper bound $\hat{\varsigma}(h')$ on $\varsigma(h')$. If the upper bound $\hat{\varsigma}(h')$ is less than the score of the exemplar $\varsigma(\hat{h})$ then the supergraph h' is pruned – otherwise it is added to the queue of graphs to consider.

sLeap [30] improves on the basic Branch-And-Bound framework by integrating a heuristic pruning condition into the Branch-And-Bound algorithm. The amount of heuristic pruning is controlled by a parameter γ (called σ in the original paper). The condition prunes the current subgraph h if it has a supergraph h' that has already been processed and the difference in support for h and h' is with-in γ. The pruning condition is heuristic, meaning sLeap only approximates the behavior of Branch-And-Bound.

LEAP Search [30] uses sLeap as a subroutine of *Frequency Descending Mining*. As this name suggests, it works by running the sLeap algorithm repeatedly, each time with a lower setting for the minimum frequency. At each step, the minimum frequency is halved until either it reaches 1 or the output of sLeap does not change between runs. Note that each iteration of LEAP feeds the output of sLeap back into itself to seed the set of maximally scored subgraphs. This pre-seeding of sLeap allows later iterations to prune the search space much faster. An (optional) last step of LEAP then runs sLeap one last time with the heuristic pruning parameter $\gamma < 0$ so that no heuristic pruning is performed.

To the author's knowledge this paper presents the first algorithm to sample significant subgraphs. However, there have been a number of studies on sampling frequent subgraphs [32], [44]–[49]. Zou and Holder [44] create a representative sample of the large graph to which they apply a traditional frequent subgraph mining algorithm. The rest of the studies [32], [45]–[49] model the search space as a frequent connected subgraph lattice and perform various random walks on the lattice in a similar fashion to SWRW. These systems intend to provide a representative sample of the frequent subgraphs in the dataset while SWRW attempts to extract the most significant subgraphs.

VIII. Conclusions

We presented Score Weighted Random Walks (SWRW), a new algorithm for Suspicious-Behavior Based Fault Localization (SBBFL). SWRW randomly samples suspicious subgraphs of dynamic control flow graphs of passing and failing executions, favoring selection of the most suspicious subgraphs. Unlike previous algorithms for SBBFL, SWRW may be used with a wide variety of suspiciousness metrics. Nine metrics were adapted from coverage based statistical fault localization. An empirical study was conducted on five real world programs written in the Go programming language. To support the study a new profiling tool for Go, Dynagrok, was developed. The results indicate that SWRW is more accurate and scalable than similar behavioral fault localization algorithms.

REFERENCES

[1] J. Jones, M. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, 2002.

[2] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, vol. 40, no. 6, p. 15, jun 2005.

[3] C. Liu, H. Yu, P. S. Yu, X. Yan, H. Yu, J. Han, and P. S. Yu, "Mining Behavior Graphs for Backtrace of Noncrashing Bugs," in *Proceedings of the 2005 SIAM International Conference on Data Mining*. Society for Industrial and Applied Mathematics, 2005, pp. 286–297.

[4] J. Jones, "Fault localization using visualization of test information," in *Proceedings. 26th International Conference on Software Engineering*, vol. 1, no. 1. IEEE Comput. Soc, 2004, pp. 54–56.

[5] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. New York, New York, USA: ACM Press, 2016, pp. 165–176.

[6] F. Tip, "A survey of program slicing techniques," *Journal of Programming Languages*, vol. 3, no. 3, pp. 121–189, 1995.

[7] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying Bug Signatures Using Discriminative Graph Mining," in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ser. ISSTA '09. New York, NY, USA: ACM, 2009, pp. 141–152.

[8] G. Di Fatta, S. Leue, and E. Stegantova, "Discriminative Pattern Mining in Software Fault Detection," in *Proceedings of the 3rd International Workshop on Software Quality Assurance*, ser. SOQUA '06. New York, NY, USA: ACM, 2006, pp. 62–69.

[9] F. Eichinger, K. Böhm, and M. Huber, *Mining Edge-Weighted Call Graphs to Localise Software Bugs*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 333–348.

[10] F. Eichinger, K. Krogmann, R. Klug, and K. Böhm, "Software-defect Localisation by Mining Dataflow-enabled Call Graphs," in *Proceedings of the 2010 European Conference on Machine Learning and Knowledge Discovery in Databases: Part I*, ser. ECML PKDD'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 425–441.

[11] Z. Mousavian, M. Vahidi-Asl, and S. Parsa, "Scalable Graph Analyzing Approach for Software Fault-localization," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: ACM, 2011, pp. 15–21.

[12] F. Eichinger, C. Oßner, and K. Böhm, "Scalable software-defect localisation by hierarchical mining of dynamic call graphs," *Proceedings of the 11th SIAM International Conference on Data Mining, SDM 2011*, no. c, pp. 723–734, 2011.

[13] S. Parsa, S. A. Naree, and N. E. Koopaei, "Software Fault Localization via Mining Execution Graphs," in *Proceedings of the 2011 International Conference on Computational Science and Its Applications - Volume Part II*, ser. ICCSA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 610–623.

[14] L. Mariani, F. Pastore, and M. Pezze, "Dynamic Analysis for Diagnosing Integration Faults," *IEEE Trans. Softw. Eng.*, vol. 37, no. 4, pp. 486–508, jul 2011.

[15] A. Yousefi and A. Wassyng, "A Call Graph Mining and Matching Based Defect Localization Technique," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, mar 2013, pp. 86–95.

[16] A. Aho, R. Sethi, M. S. Lam, and J. D. Ullman, *Compilers: principles, techniques, and tools*, 2007.

[17] 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.

[18] R. Abreu, P. Zoeteweij, and A. Van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 2006, pp. 39–46.

[19] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: Effective Statistical Debugging via Efficient Path Profiling," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 34–44.

[20] G. G. K. Baah, A. Podgurski, and M. J. M. Harrold, "Causal inference for statistical fault localization," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 73–84.

[21] G. K. Baah, A. Podgurski, and M. J. Harrold, "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.

[22] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," *Proceedings of the 2013 International Symposium on Software Testing and Analysis - ISSTA 2013*, p. 314, 2013.

[23] S. Yoo, M. Harman, and D. Clark, "Fault Localization Prioritization: Comparing Information-theoretic and Coverage-based Approaches," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 19:1—-19:29, jul 2013.

[24] Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of Software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, feb 2014.

[25] T.-D. B. Le, R. J. Oentaryo, and D. Lo, "Information retrieval and spectrum based bug localization: better together," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, no. 65. New York, New York, USA: ACM Press, 2015, pp. 579–590.

[26] S.-F. Sun and A. Podgurski, "Properties of Effective Metrics for Coverage-Based Statistical Fault Localization," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, apr 2016, pp. 124–134.

[27] T. Kudo, E. Maeda, and Y. Matsumoto, "An Application of Boosting to Graph Classification," in *Proceedings of the 17th International Conference on Neural Information Processing Systems*, ser. NIPS'04. Cambridge, MA, USA: MIT Press, 2004, pp. 729–736.

[28] H. Saigo, N. Krämer, and K. Tsuda, "Partial least squares regression for graph mining," in *Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD 08*. New York, New York, USA: ACM Press, 2008, p. 578.

[29] M. Thoma, H. Cheng, A. Gretton, J. Han, H.-P. Kriegel, A. Smola, L. Song, P. S. Yu, X. Yan, and K. M. Borgwardt, "Discriminative frequent subgraph mining with optimality guarantees," *Statistical Analysis and Data Mining*, vol. 3, no. 5, pp. 302–318, aug 2010.

[30] X. Yan, H. Cheng, J. Han, and P. S. Yu, "Mining Significant Graph Patterns by Leap Search," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 433–444.

[31] H. Cheng, X. Yan, and J. Han, "Mining Graph Patterns," in *Frequent Pattern Mining*. Cham: Springer International Publishing, 2014, pp. 307–338.

[32] T. A. D. Henderson and A. Podgurski, "Sampling Code Clones from Program Dependence Graphs with GRAPLE," in *International Workshop on Software Analytics*. ACM, 2016.

[33] J. G. Kemeny and J. L. Snell, *Finite Markov Chains*, 1st ed. Princeton, NJ: Van Nostrand, 1960.

[34] C. M. Grinstead and J. L. Snell, *Introduction to Probability*, 2nd ed. Providence, RI: American Mathematical Society, 1997.

[35] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440.

[36] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and Improving Fault Localization," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 609–620.

[37] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are Mutants a Valid Substitute for Real Faults in Software Testing?" in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: ACM, 2014, pp. 654–665.

[38] D. Lo, H. Cheng, J. Han, S.-C. Khoo, and C. Sun, "Classification of software behaviors for failure detection," in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '09*. New York, New York, USA: ACM Press, 2009, p. 557.

[39] X. Yan and J. Han, "gSpan: graph-based substructure pattern mining," in *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* IEEE Comput. Soc, 2002, pp. 721–724.

[40] ——, "CloseGraph: Mining Closed Frequent Graph Patterns," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '03. New York, NY, USA: ACM, 2003, pp. 286–295.

[41] M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen, "A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston," in *9th European Conference on Principles and Practice of Knowledge Discovery in Databases*. Porto, Portugal: Springer Berlin Heidelberg, 2005, pp. 392–403.

[42] T. Ball and J. Larus, "Efficient path profiling," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. IEEE Comput. Soc. Press, 1996, pp. 46–57.

[43] H.-Y. Hsu, J. A. Jones, and A. Orso, "Rapid: Identifying Bug Signatures to Support Debugging Activities," in *International Conference on Automated Software Engineering*. IEEE, sep 2008, pp. 439–442.

[44] R. Zou and L. B. Holder, "Frequent subgraph mining on a single large graph using sampling techniques," *Proceedings of the Eighth Workshop on Mining and Learning with Graphs - MLG '10*, pp. 171–178, 2010.

[45] V. Chaoji, M. Al Hasan, S. Salem, J. Besson, and M. J. Zaki, "ORIGAMI: A Novel and Effective Approach for Mining Representative Orthogonal Graph Patterns," *Stat. Anal. Data Min.*, vol. 1, no. 2, pp. 67–84, jun 2008.

[46] M. Al Hasan and M. J. Zaki, "Output Space Sampling for Graph Patterns," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 730–741, aug 2009.

[47] M. Al Hasan and M. Zaki, *Musk: Uniform Sampling of k-Maximal Patterns*. Philadelphia, PA: Society for Industrial and Applied Mathematics, apr 2009, ch. 55, pp. 650–661.

[48] T. K. Saha and M. A. Hasan, "FS³: A sampling based method for top-k frequent subgraph mining," in *2014 IEEE International Conference on Big Data (Big Data)*. IEEE, oct 2014, pp. 72–79.

[49] T. A. D. Henderson and A. Podgurski, "Rethinking Dependence Clones," in *International Workshop on Software Clones*. IEEE, 2017.