PROSECUTOR: Bayesian Counterfactual Fault Localization

SARA BARADARAN, University of Southern California, USA
YIFEI HUANG, University of Southern California, USA
WEI LE, Iowa State University, USA
MUKUND RAGHOTHAMAN, University of Southern California, USA

This paper presents a new approach to fault localization. We consider the introduction of errors and their subsequent propagation through faulty execution traces as a stochastic process. This allows us to perform Bayesian inference on a probabilistic model extracted from runtime program dependencies and associate individual statements and values with a posterior suspicion/belief of being erroneous. Next, we confirm or refute these beliefs by performing a series of counterfactual experiments on the program. Each experiment involves artificially flipping a suspicious branch condition at runtime and observing its downstream effect on test outcome. We have implemented our tool, which we call PROSECUTOR, and evaluated it on 470 buggy versions of 13 projects from the Defects4J benchmark suite. Our experimental evaluation shows that 40% of the true fault locations are identified within just the three lines of code believed to be the most suspicious. The technique also turns out to be significantly more effective than a diverse set of baselines, and successfully identifies at least 19%, 32%, 23%, and 18% *more* true fault locations than each of the baselines within its predictions for the top 1, 3, 5, and 10 most suspicious program statements, respectively.

ACM Reference Format:
Sara Baradaran, Yifei Huang, Wei Le, and Mukund Raghothaman. 2026. PROSECUTOR: Bayesian Counterfactual Fault Localization. 1, 1 (February 2026), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The fault localization problem may be stated as follows: Given a faulty program whose bug is witnessed by a failing test case, rank, in order of suspicion, the lines of code that are most likely to be the location of the bug. Starting with Jones and Harrold's groundbreaking work on Tarantula [10], this problem has been the subject of a rich body of research in software engineering.

Techniques range from the so-called *spectrum-based fault localization (SBFL)*—which monitor test coverage to identify parts of the program that are disproportionately executed by failing tests and infrequently executed by passing tests [3, 4, 10, 22, 29]—to *mutation-based fault localization (MBFL)*, which apply a series of tentative modifications to the program and observe their downstream effects on test outcomes [21, 23].

These approaches have complementary strengths and weaknesses: Spectrum-based methods are lightweight (they simply need some form of code coverage monitoring), but rely on access to a comprehensive suite of tests to be effective. Moreover, their reliance on aggregate statistics gives them limited discriminatory power. In contrast, mutation-based techniques provide actionable evidence for possible locations of the bug, but because of the large space of mutations and the need to repeatedly rerun the entire test suite, their scalability is limited in practice.

With this background, our paper presents a new approach to localizing faults. *First*, we view the introduction and propagation of errors through program executions as a *stochastic* process. Every statement is associated with a prior suspicion/belief of being buggy, and these buggy statements have some probability of mapping correct inputs to incorrect outputs. We model runtime program

Authors' Contact Information: Sara Baradaran, University of Southern California, Los Angeles, CA, USA, sbaradar@usc.edu; Yifei Huang, University of Southern California, Los Angeles, CA, USA, yifeih@usc.edu; Wei Le, Iowa State University, Ames, IA, USA, weile@iastate.edu; Mukund Raghothaman, University of Southern California, Los Angeles, CA, USA, raghotha@usc.edu.

2026. ACM XXXX-XXXX/2026/2-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

dependencies as a Bayesian network, which represents correlations between the correctness of various program elements, including statements and runtime values. The idea is that marginal inference on this probabilistic model, conditioned on observations from test cases, can be used to calculate the posterior suspicion that individual statements/values are erroneous.

Next, we use these posterior suspiciousness scores to identify a set of *counterfactual* experiments. Each of these experiments involves artificially changing the value of a branch predicate at runtime and observing its effect on eventual test outcome. Intuitively stated: If a branch condition was incorrectly evaluated, then it is possible that artificially flipping its value might cause the failing test to succeed. Alternatively, if a statement is buggy, then it is possible that rerouting control flow around this statement prevents test failure. These experiments help to confirm or refute our belief in the correctness/incorrectness of the branch conditions and their dependent statements, providing an additional source of information that can be incorporated into the ranking process.

In contrast to spectrum-based approaches, the fine-grained tracking of program dependencies allows us to more precisely reason about the location of faults. On the other hand, in contrast to mutation-based techniques, our approach performs a significantly smaller number of counterfactual experiments, and is able to more precisely incorporate information from their outcomes.

Due to the availability of the Soot framework [27] and Just et al.'s Defects4J dataset of real-world bugs [11], we implemented our approach to locate bugs in Java programs. The first challenge is that, because of their object-oriented nature, Java programs have complicated data and control flow. This calls for precise modeling of error propagation along execution traces, both within and across procedure boundaries. Our solution involves using a graph data structure that we call the error propagation graph (EPG) alongside an algorithm that encodes runtime program dependencies into this graph. This allows us to track errors introduced through both data and control flow, while still managing to scale to long traces.

The idea of viewing the fault localization problem through the lens of probabilistic reasoning is admittedly not new [33, 35]. Still—the key to any such approach is the probabilistic model or the underlying graph of correlations used to calculate the suspiciousness of each line of code. In this context, our main contribution is the design of the EPG. In addition to direct dependencies between the site of a variable definition and the point of its use, we also, for example, include edges from unexplored branches that contain unexecuted variable assignments. Such careful modeling is crucial, and as we will see in our experimental evaluation, our modeling contributes to a 7.5% *decrease* in EXAM score over SmartFL [35], the previous state-of-the-art probabilistic baseline, and a 41% increase in the number of bugs localized by examining just three lines of code.

The second challenge is that marginal inference is famously intractable [14]. We therefore have to suitably compress the graph of runtime program dependencies so that we can use standard inference algorithms such as loopy belief propagation [13]. Our solution involves an instrumentation-guided loop identification algorithm to identify and compress loops in execution traces.

The final question is this: What counterfactual experiments do we conduct, and how do we incorporate their results back into the probabilistic model? We restrict ourselves to the 20 most suspicious branch conditions identified by an initial Bayesian inference run. Due to the limited number of experiments, our technique is massively more scalable than MBFL. In order to leverage information from these experiments, we extend the initial probabilistic model with virtual variables and edges. This allows us to model the impact of flipping the branch condition on other dependent statements as well.

We call our implementation Prosecutor and evaluated it on faulty versions of 13 projects from the Defects4J dataset [11]. We compare Prosecutor to 8 baselines: 4 from SBFL family, 2 from MBFL family, and 2 state-of-the-art methods, SmartFL and DepGraph [24], which leverages probabilistic models and graph neural networks (GNN), respectively. We observed that Prosecutor identifies

at least 19%, 32%, and 23% more true fault locations within its top-1, 3, and 5 statement-level predictions than any of the baselines under consideration. Moreover, we observe that, unlike SBFL and MBFL, whose effectiveness strictly relies on the presence of a high-coverage test suite with both passing and failing tests, our approach can effectively localize faults without even consulting the passing traces.

In summary, this paper makes the following contributions:

(1) A systematic approach to model runtime program dependencies in object-oriented languages as Bayesian networks.
(2) A new fault localization approach with a central probabilistic model of error introduction and propagation, which leverages counterfactual analysis to reason about the location of faults.
(3) Optimizations, including loop compression and eliminating redundant dependencies, which enable scaling the technique to real-world software.
(4) Empirical evaluation on buggy versions of 13 projects from the Defects4J benchmark to showcase that PROSECUTOR outperforms existing approaches in identifying faults.

2 Motivating Example

Consider the buggy Java program in Figure 1. As shown in the assertions on Lines 32 and 35, the goal of this function is to pad the left side of the input string until it reaches the desired length. In the case of the failing test, the incorrect assignment to the `padChars` variable leads to an unexpected out-of-bounds array access on Line 26. In contrast, the passing test case never uses the erroneous variable, and correctly returns with the expected output on Line 22.

We argue that neither spectrum-based nor mutation-based approaches satisfactorily localize this bug: In particular, as can be seen from the trace annotations, Lines 24–26 are the only lines of code that are uniquely executed by the failing test, but none of them are buggy. As such, spectrum-based approaches cannot distinguish between the remaining statements. On the other hand, MBFL techniques typically focus on mutating operators rather than operands (which require careful consideration of types and contexts to safely mutate) and are therefore unable to mutate the buggy Line 9. In addition, MBFL techniques such as Metallaxis also consider the effect of the mutation on test *behavior*, rather than simply the eventual *outcome*.

```
1  public class StringUtils {
2    public static String leftPad(String str, int size, String padStr) {
3      if (str == null) {
4        return null;
5      }
6      if (padStr.length() == 0) {
7        padStr = " "; // pads with space if padStr not specified.
8      }
9      char[] padChars = str.toCharArray(); // faulty line
10     int padLen = padStr.length();
11     int strLen = str.length();
12     int pads = size - strLen;
13     if (pads <= 0) {
14       return str; // returns original String when possible
15     }
16     if (padLen == 1 && pads <= PAD_LIMIT) {
17       return leftPad(str, size, padStr.charAt(0));
18     }
19     if (pads == padLen) {
20       return padStr.concat(str);
21     } else if (pads < padLen) {
22       return padStr.substring(0, pads).concat(str);
23     }
24     char[] padding = new char[pads];
25     for (int i = 0; i < pads; i++) {
26       padding[i] = padChars[i % padLen]; // crash
27     }
28     return new String(padding).concat(str);
29   }
30 }
31 public void testCase1() { // an example of passing test
32   assertEquals("!aabc", StringUtils.leftPad("abc", 5, "!aL"));
33 }
34 public void testCase2() { // an example of failing test
35   assertEquals("12341234a", StringUtils.leftPad("a", 9, "1234"));
36 }
```

Fig. 1. Example implementation of the `leftPad` utility function, adapted from the Apache Commons Lang library [1]. The program contains a bug on Line 9, which should instead be `padChars = padStr.toCharArray()`. Graph annotations on the right side describe the execution flow in each test case.

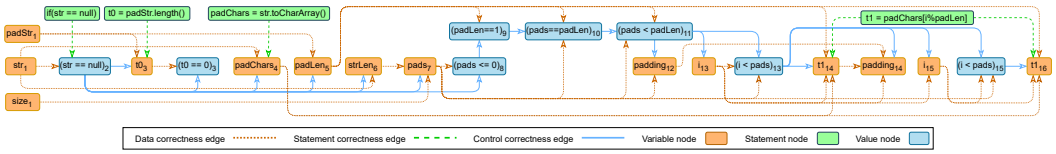

Fig. 2. Snippet of the error propagation graph (EPG) for the trace obtained from `testCase2` in Figure 1.

In this context, mutating the `if`-statements on Lines 3, 6, 13, 16, 19, and 21 may provide opportunities for premature return, thereby eliminating the crash on Line 26, and instead causing an assertion violation at Line 35. The algorithm therefore characterizes them as "successful" mutations, despite being irrelevant to the bug, leading to an unsatisfactory ranking of possible fault locations.

2.1 Postmortem of the Fault

Line 26 may be regarded as consisting of two operations: the first which loads and computes the value `padChars[i % padLen]`, and the second which stores this value in `padding[i]`. It is the first operation, i.e., the computation of `padChars[i % padLen]`, which results in the crash. By examining the source code and crash context, we conclude that one of the following *must* be the case: (*a*) the load operation on Line 26 is faulty, or (*b*) one of the operand variables, `padChars`, `i` and `padLen`, was incorrectly computed, or that (*c*) control must never have reached Line 26, forcing us to examine whether the loop guard on Line 25 was correctly evaluated in the last iteration of this loop. This backward analysis of program dependencies leads us to suspect and examine four lines of code: Lines 9, 10, 25, and 26. Much of our paper follows from the following more general proposition:

> Consider the sequence of statements, $t_1, t_2, \ldots, t_n$ executed along a program trace. If executing a statement t_i introduces or propagates an error into the program state, then either: (*a*) t_i is itself buggy, or (*b*) one of its operands was incorrectly computed, or (*c*) the branch condition that controls the execution of t_i was incorrectly evaluated.

Our approach leverages program dependencies to construct a model of error propagation instead of just using coarse-grained coverage information. We encode the dependencies between the correctness / incorrectness of runtime values using a data structure that we call the *error propagation graph (EPG)*. A fragment of such a graph is illustrated in Figure 2.

At a high level, the error propagation graph contains three kinds of nodes: statement nodes, value nodes, and variable nodes respectively. Each statement node represents the correctness of the corresponding program statement, while variable nodes represent the correctness of the variables defined along the program trace. We also associate statements which do not define new variables but rather only evaluate expressions in order to change control flow, with so-called value nodes corresponding to the correctness of the expressions evaluated by these statements at runtime.

For example, the node $\texttt{padChars}_4$ in the graph of Figure 2 indicates that in the failing trace, the correctness of the value assigned to `padChars` in Step 4 depends on the correctness of the statement `padChars = str.toCharArray()` in Line 9 and the value of `str` defined at the method's entry point. It also depends on whether the branch condition at Step 2 is evaluated correctly. Furthermore, the node illustrates that an error which reaches the execution Step 4 and taints the value of variable `padChars` may subsequently propagate and affect the value of the temporary variable `t1` in Steps 14 and 16. For the purpose of exposition and to avoid cluttering the graph, we only show a part of statement nodes which affect values such as $\texttt{padChars}_4$, $\texttt{t1}_{14}$, and $\texttt{t1}_{16}$. Note however, that the complete EPG contains similar nodes for *all* other runtime values.

Another variable node of interest, $\texttt{t1}_{16}$, corresponds to the value temporarily created by the load operation at Step 16, *had it not crashed*. Following the earlier discussion above, the correctness of this variable depends on the correctness of the load statement, $\texttt{padChars}_4$, $\texttt{padLen}_5$, and $\texttt{i}_{15}$. It also depends on whether the branch condition `(i < pads)` at Step 15 was evaluated correctly.

Observe that the EPG effectively constrains the manner in which errors may be introduced or propagated along the execution trace. If a statement is incorrect, then forward analysis from the corresponding statement node will uncover all values and variables that might have been incorrectly computed. Conversely, if a value is found to be incorrect, then the erroneous statement must reside in the backward cone of influence from this node in the EPG.

Observe also that the subgraph corresponding to the value and variable nodes resembles the program dependence graph (PDG), which is constructed by integrating control and data dependencies. The key difference between conventional PDGs and this subgraph of the EPG is that the nodes in PDGs are program statements and the edges represent control and data dependencies between them. In contrast, the blue and yellow nodes in Figure 2 depict runtime values from each step of the execution trace and the edges represent correctness dependencies between them. Also, note that the EPG can be generalized to the setting of having multiple test executions. In these situations, the subgraphs corresponding to different test executions would share the same set of statement nodes, but would have their own separate sets of value and variable nodes, representing the runtime behavior of the corresponding execution.

2.2 The Probabilistic Model

When faced with buggy code, software engineers are initially unsure of the exact location of the fault, but might be suspicious of different program statements. Another key aspect of the error propagation graph is that it enables us to quantitatively reason about correlations between the correctness of various program elements.

For example, consider the node $\texttt{padLen}_5$. If all of its predecessors in the EPG were correctly evaluated, then this node must also be correctly computed. I.e.,

$$\Pr(\texttt{padLen}_5 \mid h_0) = 1,$$

where $h_0 = \texttt{padStr}_1 \wedge (\texttt{str} == \texttt{null})_2 \wedge \texttt{padLen} = \texttt{padStr.length()}$ corresponds to the event that none of its predecessors were incorrect. On the other hand, this variable might also have been incorrectly computed for multiple reasons. If, for example, an incorrect input string `padStr` was provided, then we declare:

$$\Pr(\texttt{padLen}_5 \mid h_1) = 0.85,$$

where $h_1 = \neg\texttt{padStr}_1 \wedge (\texttt{str} == \texttt{null})_2 \wedge \texttt{padLen} = \texttt{padStr.length()}$. Notice that the statement has some degree of "*fault tolerance*": i.e., even if `padStr` was incorrect, it might be assigned a string which accidentally has the correct length, so that the $\texttt{padLen}_5$ variable is nevertheless correct. In this manner, we associate each vertex in the EPG with a conditional probability distribution (CPD), which describes our belief that it was correctly evaluated, conditioned on the correctness status of its immediate predecessors.

Taken together, these CPDs allow us to treat the EPG as a Bayesian network, where each vertex is a Boolean-valued random variable. We also know from executing the tests that some variables in the EPG were incorrect as the test failed, and that some others were (presumably) correct as the test succeeded. This allows us to run the Bayesian network in "*reverse*" and recover the marginal probability $\Pr(\neg l \mid e)$ that each statement l is buggy, conditioned on our observations e on test behaviors. We show the illustrative results of these calculations in Table 1.

Table 1. The ranked list of suspiciousness scores reported by PROSECUTOR for the 10 most suspicious program statements in Figure 1. This is obtained by calculating posterior probability $\Pr(\neg l \mid \neg \texttt{t1}_{16})$ for each statement l, given the EPG in Figure 2 as a Bayesian network and the observation $\neg \texttt{t1}_{16}$ on the failing execution.

Rank	Suspicion	Line:Statement
1	0.531819	26:t1 = padChars[i % padLen]
2	0.261497	25:if(i < pads)
3	0.205962	9:padChars = str.toCharArray()
4	0.172209	25:i++
5	0.168113	25:i = 0
6	0.166703	12:pads = size - strLen
7	0.162269	21:if(pads < padLen)
8	0.153250	3:if(str == null)
9	0.152615	10:padLen = padStr.length()
10	0.150381	11:strLen = str.length()

These suspiciousness scores allow us to naturally rank individual program statements for triage by developers. For the program of Figure 1, one needs to inspect just 3 lines of code to discover the true fault location. More broadly, in our evaluation in Section 5, we find that developers would discover 40% of bugs if they similarly examined only the first 3 statements reported by PROSECUTOR.

2.3 Counterfactual Executions

Consider a second scenario, as shown in Figure 3, where the bug now exists in Line 22. This is a common pitfall: programmers may mistakenly use the wrong method when multiple overloaded methods share the same name. Here, the previously passing test, `testCase1`, now fails since an incorrect string `"L"` is returned by the statement `padStr.substring(pads)` in Line 22. Notice that during the execution of `testCase1`, if we flip the condition value in Line 21 at Step 11, then the faulty statement would not be executed and, followed by executing Lines 24–28, the test passes successfully. This may provide evidence about either the incorrectness of the condition value $(\texttt{pads} < \texttt{padLen})_{11}$ in the original failing execution or its dependent variables/values. We can thus leverage such observations by incorporating them into the ranking process.

```
2   public static String leftPad(String str, int size, String padStr) {
3     ...                                                                  2
9     char[] padChars = padStr.toCharArray(); // fixed                     4
10    ...                                                                  :
21    } else if (pads < padLen) {                                          11
22        return padStr.substring(pads).concat(str); // faulty line        12
23    }
24    char[] padding = new char[pads];
25    for (int i = 0; i < pads; i++) {
26      padding[i] = padChars[i % padLen];
27    }
28    return new String(padding).concat(str);

31  public void testCase1() { // Now, an example of failing test
32    assertEquals("!aabc", StringUtils.leftPad("abc", 5, "!aL"));          1 13
33  }
```

Fig. 3. A variant of the code in Figure 1 with a bug on Line 22, which should instead call `padStr.substring(0, pads)`.

Similar to the graph in Figure 2, the EPG can be constructed for the failing execution obtained from `testCase1` in Figure 3. Given that an assertion violation occurs on Line 32, by performing marginal inference on the constructed graph, we calculate the posterior probabilities $\Pr(\neg t_i \mid \neg \texttt{assert}_{13})$ for each branch condition t_i in EPG (blue nodes in Figure 2). This procedure creates a ranked list of branch conditions that are suspected to be incorrectly evaluated. Next, for each condition in the top 20 entries of the ranking, we artificially flip it at runtime and observe its downstream effect on the test outcome. If a previously failing test now passes in this counterfactual scenario, then it might be the case that in the original failing execution (a) this branch condition was evaluated incorrectly, or (b) a value/variable was incorrectly computed under the control of this branch condition. E.g., flipping the branch predicate `(pads < padLen)` at Step 11 changes the test behavior since the value calculated and returned at Step 12 was erroneous.

In such cases, we create a new virtual node, named `cfx`, in the EPG with incoming edges from branch values, flipping of which changes the test outcome and their immediate successors. Finally, we calculate the posterior probability $\Pr(\neg l \mid e \wedge \neg \texttt{cfx})$ for each statement l as its suspiciousness

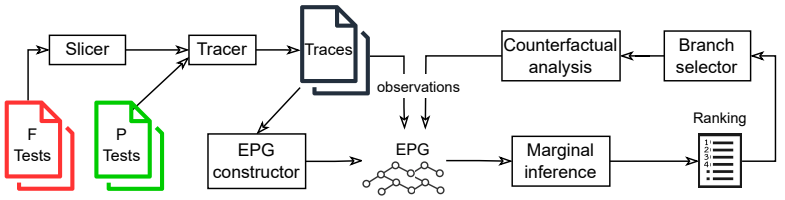

Fig. 4. The PROSECUTOR fault localization workflow.

score. Note that treating `cfx` as an erroneous variable enables the model to consider that either nodes with incoming edges to `cfx` could be incorrect, and this suspicion will be further propagated to all other nodes reachable from `cfx`.

Concretely, feeding the the above counterfactual results back into the model will raises the suspiciousness score of Line 22 from 0.47 to 0.61 and places this line at the top of the ranking.

3 Methodology

Figure 4 presents a high-level overview of the PROSECUTOR fault localization workflow. We now describe this workflow in detail, including the EPG construction process (Section 3.1), a recap of Bayesian networks and the manner in which we probabilistically view the EPG (Section 3.2). Finally, the process of collecting observations, performing counterfactual analysis and computing suspiciousness scores will be elaborated (Section 3.3).

3.1 Constructing the Error Propagation Graph

The EPG is fundamentally constructed from runtime control and data dependencies. We precompute the set of static control dependencies along with DEF/USE sets through a static analysis phase.

3.1.1 Static Program Dependence Analysis. To compute data dependencies in object-oriented languages like Java, we need to consider objects as complex variables with state that can change through manipulation of their attributes. Consider a field-insensitive analysis where changing any attribute of an object changes the entire state of the object. This is straightforward when restricted to the scope of a procedure. One can add the object to the DEF set upon encountering a statement which updates one of its fields. However, it becomes tricky to track object modifications across procedure boundaries and propagate their effects on data dependencies to the caller.

Instead of being fully precise in tracking object modifications, we efficiently approximate the set of defined objects at call sites. We consider that the state of an object u may change when (a) one of the methods of u is called, or (b) a reference to u is passed as an argument to another procedure. We therefore suggest adding receiver objects to the DEF set at call sites as well as the objects and other reference variables, including arrays, which are passed as arguments in call statements. This allows the local def-use chain computation to assume that the new versions of these variables are used in the caller after returning from the callee.

This over-approximation of defined objects might be inefficient due to introducing several spurious data dependencies. Even though invoking a method of an object can change its attributes and therefore the object's state, we assert that this possibility is not identical across different invocations. Invoking *setter-like* methods, which do not return values, including constructors, is very likely to change the state of an object, while *getter-like* methods are often designed to perform computations and eventually return the results. We thus make a more conservative approximation by adding receiver objects to the DEF set only in the case of calling methods with `void` return type. Based on this view of data dependency, we specify DEF/USE sets for primitive statements in Table 2.

Table 2. Defined and used variables in 3-address statements. R_a is the set of reference variables passed as arguments to the callee procedure. The main difference with classical Def/Use analysis is in method calls, where we treat both receiver objects and reference arguments as being possibly defined.

Stmt type	Stmt format	Def	Use
Assign Stmt	t = new Class(a_1, ..., a_n)	$\{t\} \cup R_a$	$\{a_1, \ldots, a_n\}$
	t = new Type[x]	$\{t\}$	$\{x\}$
	t = u.field	$\{t\}$	$\{u\}$
	u.field = t	$\{u\}$	$\{u, t\}$
	t = u[x]	$\{t\}$	$\{u, x\}$
	u[x] = t	$\{u\}$	$\{u, x, t\}$
	t = u	$\{t\}$	$\{u\}$
	t = u_1 binop u_2	$\{t\}$	$\{u_1, u_2\}$
Return Stmt	return t	$\varnothing$	$\{t\}$
	return	$\varnothing$	$\varnothing$
If Stmt	if u_1 binop u_2 goto L	$\varnothing$	$\{u_1, u_2\}$
Goto Stmt	goto L	$\varnothing$	$\varnothing$
Throw Stmt	throw t	$\varnothing$	$\{t\}$
Catch Stmt	catch e	$\{e\}$	$\varnothing$
Invoke Stmt	t = u.method(a_1, ..., a_n)	$\{t\} \cup R_a$	$\{a_1, \ldots, a_n, u\}$
	u.method(a_1, ..., a_n)	$\{u\} \cup R_a$	$\{a_1, \ldots, a_n, u\}$
	t = function(a_1, ..., a_n)	$\{t\} \cup R_a$	$\{a_1, \ldots, a_n\}$
	function(a_1, ..., a_n)	R_a	$\{a_1, \ldots, a_n\}$

To obtain control dependencies, we statically calculate dominance frontiers on the reverse control-flow graph (CFG) of each procedure (Rdf relations), as proposed by Cytron et al. [8]. This approach determines control dependencies introduced by conventional control-flow constructs (e.g., `if-else`) with explicit edges in CFG. On the other hand, Java programs also make frequent use of `try`, `catch`, and `throw` constructs, which introduce implicit edges from `throw` sites to `catch` statements. For instance, in the code fragment `try {...} catch(Exception e){...}`, several statements inside the `try` block may potentially throw an exception, which would cause control to jump directly to the `catch` statement. We thus assume that every statement in a `catch` block is statically control-dependent on the corresponding `catch` statement, which itself is control-dependent on all statements in the associated `try` block.

3.1.2 Online Graph Construction. We now explain the online process of computing runtime dependencies and constructing the EPG. To specify the EPG, we need to define the set of its vertices and edges. Consider a given trace $\tau = t_1, t_2, \ldots, t_n$ in which every statement $t_i \in \tau$ is an instance of a program statement, $l_i = \textsc{Stmt}(t_i)$.

Vertices, V. We include all these program statements in the set of *statement* vertices: $V_l = \{l_i \mid t_i \in \tau\}$. Next, we associate every statement $t_i \in \tau$ with a set of vertices V_i. For every variable $v \in \textsc{Def}(l_i)$ defined at t_i, we create a *variable* vertex $\langle v@i \rangle$. If t_i is an object creation or invocation statement, then we also create a variable vertex $\langle x@i \rangle$ for each formal argument, $x \in \textsc{FArgs}(t_i)$ of its callee function. On the other hand, if t_i does not define any variables, i.e., if $\textsc{Def}(l_i) = \varnothing$, then we create a dummy *value* vertex, $\langle \#@i \rangle$. We put these together by defining $V_i = V_i^d \cup V_i^a$, where

$$V_i^d = \begin{cases} \{\langle \#@i \rangle\} & \text{if } \textsc{Def}(l_i) = \varnothing, \text{ and} \\ \{\langle v@i \rangle \mid v \in \textsc{Def}(l_i)\} & \text{otherwise,} \end{cases}$$

and $V_i^a = \{\langle x@i \rangle \mid x \in \textsc{FArgs}(t_i)\} \cup \{\langle \texttt{this}@i \rangle\}$, respectively. Intuitively, V_i^d corresponds to the variable definitions in the procedure where t_i itself is executed, while V_i^a corresponds to definitions of callee arguments created by the call statement t_i upon invocation. We collect the set of value

$$R_l \ \frac{v \in V_i}{l_i \to v \in E} \qquad R_{\text{DU}} \ \frac{v \in V_i^d \quad u \in \textsc{Use}(l_i) \quad \langle u@j \rangle = \textsc{LastDef}(u, t_i)}{\langle u@j \rangle \to v \in E} \qquad R_{\text{CTR1}} \ \frac{v \in V_i \quad t_j = \textsc{LastCtrl}(t_i) \quad u \in V_j^d}{u \to v \in E}$$

$$R_{\text{CTR2}} \ \frac{v \in V_i^d \quad u \in \textsc{Use}(l_i) \quad \langle u@j \rangle = \textsc{LastDef}(u, t_i) \quad j < r < i \quad l \in \textsc{RdfInverse}(l_r) \quad u \in \textsc{Def}(l)}{\langle \#@r \rangle \to v \in E}$$

Fig. 5. Inference rules for deriving intra-procedural EPG edges.

and variable vertices: $V_\tau = \bigcup_i V_i$, and define the final set of EPG vertices as:

$$V = V_l \cup V_\tau. \tag{1}$$

Intra-procedural edges. We now define the set of edges E in the EPG. In our implementation, we construct E by making one forward pass over the trace τ. For ease of presentation, we instead use the set of inference rules shown in Figures 5, 7, and 8 to construct E. We start by discussing the rules in Figure 5.

The Rule R_l captures the conceptually simplest set of edges: For every trace statement t_i, we include an edge from l_i to each vertex $v \in V_i$. This indicates that the correctness of v depends on the correctness of l_i. Notice that the correctness of vertices $v \in V_i^d$ also depends on the correctness of the operands u that are used. Let c_k refer to the context of a procedure invocation which executes the statement t_k. We write $\textsc{LastDef}(u, t_i)$ to refer to the vertex corresponding to the most recent definition of u that reaches the statement t_i within its context c_i. Rule R_{DU} now creates an edge from $\langle u@j \rangle$ to v, where t_j is the statement creating this variable vertex. These edges correspond to the intra-procedural def-use chains shown in yellow in Figure 2.

Finally, the correctness of each vertex $v \in V_i$ also depends on whether control should have reached this point. At each step i in the trace τ, let $\textsc{LastCtrl}(t_i)$ be the most recent statement t_j executed in the same context as t_i and which controls the execution of t_i, i.e., l_i is control-dependent on l_j. In this circumstance, Rule R_{CTR1} enables us to create an edge to v from u for each $u \in V_j^d$. These edges correspond to the intra-procedural control dependence edges shown in blue in Figure 2. Also, note that—regardless of our static over-approximation—this rule only creates edges from the actual location of the `throw` to the corresponding `catch` statement.

Now, recall the assignment `padLen = padStr.length()` in Step 5 (Line 10) of the failing trace in Figure 1. Clearly, this statement is not control dependent on the `if`-statement `if(padStr.length()== 0)` on Line 6. Nevertheless, if Line 6 was erroneous, then it might have resulted in a failure to update `padStr` on Line 7, thereby resulting in an incorrect operand being used in Step 5 of the failing trace.

Abstractly: The classical definition of control dependency assumes that a statement l is control-dependent on another statement l' if the outcome of l' determines whether l is executed. This definition does not consider the impact of this outcome on the values on l. To account for the effect of such unexplored branches, we introduce new dependency edges into the EPG using Rule R_{CTR2}.

Let $\textsc{RdfInverse}(l_r)$ be the set of all statements that are statically control-dependent on the statement l_r. Consider a statement t_i and one of its operands u. Say that u was last defined at statement t_j in the trace. Now consider all branching statements t_r, executed between t_j and t_i, such that t_r controls the execution of a fourth statement l which also defines u. In this case, Rule R_{CTR2} adds an edge from $\langle \#@r \rangle$ to v.

Example 3.1. Notice that in Figure 1, the value of the `padStr` operand in Step 5 comes from its definition in the `leftPad` entry point, specified upon invocation at Step 1. Also, the statement `if(padStr.length()== 0)` is executed at Step 3, between the definition and the use location of `padStr`. Since the defining statement `padStr = ""` on Line 7 is control-dependent on the statement

$$R_{\text{OP1}}\ \frac{\textsc{Open}_i \quad u \in \textsc{Use}(l_i) \quad \textsc{IsArg}_i(u) \quad \langle u@j\rangle = \textsc{LastDef}(u, t_i) \quad w \in \textsc{Map}(u, l_i)}{\langle u@j\rangle \rightarrow \langle w@i\rangle \in E}$$

$$R_{\text{OP2}}\ \frac{\textsc{Open}_i \quad u \in \textsc{Use}(l_i) \quad \textsc{IsRcvObj}_i(u) \quad \langle u@j\rangle = \textsc{LastDef}(u, t_i)}{\langle u@j\rangle \rightarrow \langle \texttt{this}@i\rangle \in E}$$

$$R_{\text{OP3}}\ \frac{\textsc{Open}_i \quad u \in \textsc{Use}(l_i) \quad \textsc{IsArg}_i(u) \quad \langle u@j\rangle = \textsc{LastDef}(u, t_i) \quad j < r < i \quad l \in \textsc{RdfInverse}(l_r) \quad u \in \textsc{Def}(l) \quad w \in \textsc{Map}(u, l_i)}{\langle \#@r\rangle \rightarrow \langle w@i\rangle \in E}$$

$$R_{\text{OP4}}\ \frac{\textsc{Open}_i \quad u \in \textsc{Use}(l_i) \quad \textsc{IsRcvObj}_i(u) \quad \langle u@j\rangle = \textsc{LastDef}(u, t_i) \quad j < r < i \quad l \in \textsc{RdfInverse}(l_r) \quad u \in \textsc{Def}(l)}{\langle \#@r\rangle \rightarrow \langle \texttt{this}@i\rangle \in E}$$

Fig. 7. Inference rules for deriving inter-procedural EPG edges from call-sites.

`if(padStr.length()== 0)` at Line 6, Rule R_{CTR2} derives an edge between nodes $\texttt{(t0==0)}_3$ and $\texttt{padLen}_4$ in Figure 2.

The same challenge arises when processing dependencies within syntactically infinite loops. Consider the program in Figure 6 with a `while(true)` loop. Since the `return` statement on Line 6 post-dominates the `if`-statement on Line 5, the former is not statically control-dependent on the latter. However, the correctness of the returned variable `sum` in Step 7 depends on whether the condition in Step 6 was evaluated correctly. To address this challenge, we insert a virtual exit within these loops, such as in Figure 6. Although the `throw` statement, inserted by instrumentation, is unreachable at runtime, it makes a path from Line 5 to a second exit point in the static CFG, and tricks the algorithm into establishing a control dependency between Line 5 and Line 6.

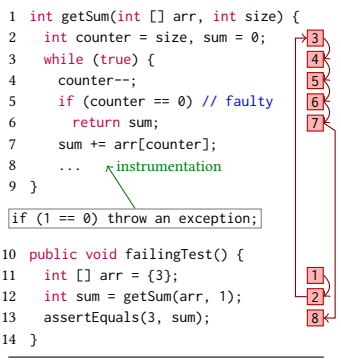

Fig. 6. An example program with a bug on Line 5, which should instead be `if (counter < 0)`.

Inter-procedural edges. So far, our model considers correctness dependencies for each procedure in isolation. This allows us to track the propagation of errors across different program elements within each context. Nevertheless, further *inter-procedural* dependencies are required to enable tracking errors across procedure boundaries. We generally need to specify: (*a*) how the correctness of elements in the caller affects the ones in the callee at the invocation site, and conversely (*b*) how the correctness of elements in the callee may affect the caller variables/values at the return site. We discuss these rules in Figures 7 and 8 respectively.

Rules R_{OP1} and R_{OP2} involve deriving inter-procedural data dependence edges that specify (*a*) in EPG. Let $\textsc{Open}_i$ be a boolean variable denoting whether the execution of the statement t_i in the trace is followed by the start of a new context. Also, let $\textsc{IsArg}_i(u)$ and $\textsc{IsRcvObj}_i(u)$ denote weather the variable u is passed as an argument or the receiver object in statement t_i, respectively. Now, for each operand u, which is last defined by the statement t_j and is passed as the argument w in t_i, Rule R_{OP1} derives an edge from $\langle u@j\rangle$ to $\langle w@i\rangle$. A similar edge is created by Rule R_{OP2} for the receiver object u.

Rules R_{OP3} and R_{OP4} are the inter-procedural counterparts of R_{CTR2} and capture the effect of unexplored branches on the correctness of argument vertices.

Note 3.2. Notice that the previous Rule R_{CTR1} also derives another set of inter-procedural control dependence edges from $u \in V_j^d$ to argument vertices $v \in V_i^a$ created upon invocation, where t_j is the statement which controls the execution of call statement t_i.

$$R_{\text{CL1}}\ \frac{\textsc{Close}_i \quad t_j = \textsc{StkTop}_i \quad v \in V_i^d \quad u \in V_j^d}{v \rightarrow u \in E}$$

$$R_{\text{CL2}}\ \frac{\textsc{Close}_i \quad t_r = \textsc{StkTop}_i \quad v \in \textsc{Def}(l_r) \quad \textsc{IsArg}_r(v) \quad w \in \textsc{Map}(v, l_r) \quad \langle w@j \rangle = \textsc{LastDef}(w, t_i) \quad r \neq j}{\langle w@j \rangle \rightarrow \langle v@r \rangle \in E}$$

$$R_{\text{CL3}}\ \frac{\textsc{Close}_i \quad t_r = \textsc{StkTop}_i \quad v \in \textsc{Def}(l_r) \quad \textsc{IsRcvObj}_r(v) \quad \langle \texttt{this}@j \rangle = \textsc{LastDef}(\texttt{this}, t_i) \quad r \neq j}{\langle \texttt{this}@j \rangle \rightarrow \langle v@r \rangle \in E}$$

Fig. 8. Inference rules for deriving inter-procedural EPG edges from return locations.

The rules in Figure 7 alongside R_{CTR1} therefore fully capture inter-procedural control and data dependencies which are necessary for (*a*).

On the other hand, inter-procedural edges that specify (*b*) are derived by Rules R_{CL1}–R_{CL3}. These rules activate when a context is closed. To calculate these edges, we need to first process all the executed statements within the callee context. Let the boolean variable $\textsc{Close}_i$ denote whether the statement t_i is the last executed statement within the context c_i. Usually, the context closes correspond to return statements. Note, however, that they capture a broader class of program statements, including crash-inducing and exception-throwing ones. Note also that when a function invocation does not gracefully return, the call statement itself serves as the point of context close. Also, let $\textsc{StkTop}_i$ denote the latest statement t_j executed in the trace before starting the context c_i. These variables allow us to link two consecutive call frames / contexts within the call stack.

Recall once again the discussion of how we encode the potential impact of unexplored branches as dependency edges within the EPG. We apply a similar reasoning to the point where the context is closed: In particular, context closing events (e.g., return) might prevent subsequent desirable updates from being applied to reference arguments of the caller, or they might return incorrect values which affect the correctness of variables storing them in the caller. Rule R_{CL1} thus preemptively creates dependency edges $v \rightarrow u$ from the return statement v to all variable vertices u created at the call site. Let t_r be the call statement. Rule R_{CL2} creates an edge from $\langle w@j \rangle$ to $\langle v@r \rangle$ for each reference variable v that is passed as an argument w, and where t_j is the statement which last defined w in the callee context. Rule R_{CL3} defines similar edges for the receiver object.

3.2 A Probabilistic View of the EPG

Conditional probability distributions. Each vertex in the EPG, $G = (V, E)$, represents the correctness of some program element, i.e., either a statement, control flow decision, or a runtime value. As previously discussed in Section 2, the edge set E encodes possible ways in which errors may propagate along the program execution.

The following principle is fundamental to the Prosecutor workflow: If all of the predecessors u of a node v in the EPG are correctly evaluated, then v is itself correctly evaluated. Conversely, if any of its predecessors u is incorrect, then v *might* itself be incorrect.

We quantify this idea by treating each program element $v \in V$ of the EPG as a Boolean-valued random variable and associating it with a conditional probability distribution (CPD). We write $\text{Pred}(v) = \{u \in V \mid u \rightarrow v \in E\}$ for the set of predecessors of v in the EPG. The conditional probability distribution of v is a function P which maps a valuation $\mathbf{x}_{\text{Pred}(v)}$ of the predecessors of v to the probability that v is true ("*correctly evaluated*"). We define:

$$P(v \mid \mathbf{x}_{\text{Pred}(v)}) = p^w, \tag{2}$$

where $p = 0.85$ and $w = \#_{\text{false}}(\mathbf{x}_{\text{Pred}(v)})$ is the number of predecessors u that were incorrectly evaluated. Conditional probabilities must add up to 1, so we define $P(\neg v \mid \mathbf{x}_{\text{Pred}(v)}) = 1 - p^w$.

Notice that when all predecessors were correctly evaluated, $w = 0$, so that $P(v \mid \mathbf{x}_{\text{Pred}(v)}) = 1$. Notice also that the conditional probability of v drops as w increases (but never reaches 0, to account for the possibility of fault-tolerant program statements). For vertices v with no incoming edges in the EPG (e.g., statement vertices $l_i \in V_l$), we uniformly use the prior probability, $\Pr(v) = 0.85$.

Bayesian networks. Note that our ultimate goal is to calculate posterior suspicions, $\Pr(\neg l \mid e)$, that each program statement l is incorrect, given our observations e on the test outcomes. In order to speak meaningfully about these conditional probabilities, we need to set up a joint probability distribution over the variables of V.

At this point, we note that, by construction, the EPG is an acyclic graph. As a result, together with the CPDs $P(v \mid \mathbf{x}_{\text{Pred}(v)})$ just defined, it forms a Bayesian network [16]. Accordingly, let $\mathbf{x}$ be a valuation of each variable in V. We define:

$$\Pr(\mathbf{x}) = \prod_{v} P(x_v \mid \mathbf{x}_{\text{Pred}(v)}). \tag{3}$$

It is clearly the case that $\Pr(\mathbf{x}) \geq 0$ and it can be verified that $\sum_{\mathbf{x}} \Pr(\mathbf{x}) = 1$. It follows that $\Pr(\mathbf{x})$ is a well-defined probability distribution. With this definition in hand, one can readily speak of the probability of individual events, $\Pr(e) = \sum_{\mathbf{x} \sim e} \Pr(\mathbf{x})$, as the sum of the probabilities of matching valuations e, and of conditional probabilities, $\Pr(e_1 \mid e_2) = \Pr(e_1 \wedge e_2)/\Pr(e_2)$. In practice, these can be computed by off-the-shelf engines such as LibDAI [20].

3.3 Evidence Collection

To calculate posterior suspicion of individual elements, we collect evidence about the correctness/incorrectness of runtime values/variables from: (*a*) Passing and failing executions on the original program, e.g., a failed assertion shows the expression inside was evaluated to false, and (*b*) passing executions on mutated versions of the program, e.g., if flipping a branch predicate results in a failing test to succeed, then this condition was likely to be evaluated incorrectly in the original execution.

We consider the trace $\tau = t_1, t_2, \ldots, t_n$ obtained upon running each test case. In the case of a failing execution, we mark variables and values $v \in V_n$ created at the *failure point* of the trace t_n as being incorrect. In passing executions, on the other hand, we mark *all* variables and values created throughout the trace, $v \in \bigcup_i V_i$ as being correctly evaluated. Although this would lower the suspiciousness $\Pr(\neg l \mid e)$ of program statements l executed in passing traces, recall that each statement still has a non-zero chance of being buggy, because of the fault tolerance built into our choice of CPDs. Furthermore, we assume that all the statements $\{l_i = \textsc{Stmt}(t_i) | t_i \in \tau\}$ inside the test procedure are bug-free and mark these statement vertices as being correct.

Counterfactual analysis. We augment our evidence about the incorrectness of program elements by performing a set of counterfactual experiments. The key idea is that if changing a value at runtime can make a failing test succeed, then this value is likely to be related to the bug. In particular, we focus on changing branch conditions as their domain has only two values, `false` and `true`.

After running the original failing test cases, we perform an initial Bayesian inference run to identify the $k = 20$ most suspicious runtime branch conditions. For each of these branch conditions t_r, in sequence: We surgically alter the program to flip the value of each specific branch condition at runtime, and rerun the failing test to determine whether this alteration causes the test to instead succeed. If the test outcome is changed, it might be the case that in the original trace: (*a*) this branch condition—specifically the value vertex $\langle \#@r \rangle$—was evaluated incorrectly, or (*b*) this condition results in executing a statement t_k that introduces or propagates an error into the program state. We account for both possibilities by introducing a new vertex `cfx` into the EPG. We also add the

following edges to E: (a) $\langle \#@r \rangle \rightarrow \texttt{cfx}$, and ($b$) $v \rightarrow \texttt{cfx}$ for each vertex v that is an immediate successor of $\langle \#@r \rangle$. We then mark the virtual variable cfx as being incorrectly evaluated.

There are two principal advantages of our approach over traditional mutation-based approaches: First, MBFL does not consider the effect of modifying a statement on its dependent statements. In other words, they only alter suspiciousness of the mutated statement in response to a successful mutation. Second, MBFL is not equipped with a built-in heuristic to restrict the large space of mutations. This imposes substantial overhead to run hundreds of experiments, which requires a huge time budget and drastically restricts the scalability of these techniques.

4 Implementation

We have implemented PROSECUTOR using the Soot framework [27], which translates Java bytecode into typed 3-address Jimple code. Our implementation consists of ≈ 3000 lines of Java to perform instrumentation and static analysis, and 2200 lines of Python code to construct the EPG. We also use LibDAI for marginal inference [20].

As shown in Figure 4, PROSECUTOR captures the last statement t_n executed in each failing test and extracts an executable backward slice, $\bigcup_{u \in \text{USE}(l_n)} \text{BACKSLICE}\langle l_n, u \rangle$, which transitively records all statements affecting a used variable u at statement l_n [6]. This allows us to have near-minimal failure-inducing tests, which reduce the size of traces by preemptively eliminating several useless paths in EPG. To construct the EPG, we use all traces F obtained from the sliced versions of the failing tests. In order to speed up EPG construction and reduce the size of the final graph, we perform a coarse-grained coverage analysis, and heuristically choose a set of passing traces P that have large overlap in their executed methods with F.

Rank aggregation. Intuitively, the traces in P provide statistical coverage information similar to SBFL. Although this information is useful, sometimes passing traces frequently execute faulty statements without themselves failing. This leads to cases where the suspiciousness of faulty statements is incorrectly suppressed, in a manner similar to the shortcomings of SBFL. We therefore construct the EPG and perform marginal inference in two settings: (a) using only failing traces in F, and (b) using traces in $F \cup P$. Somewhat amazingly, as we will see in Section 5, running PROSECUTOR in mode F already outperforms all baselines, and $F \cup P$ leads to additional improvements. Ultimately, PROSECUTOR combines the two rankings, F and $F \cup P$, by ranking statements according to $r_l = \min(r_l^{F \cup P}, r_l^F)$ and by giving priority to $F \cup P$ to break ties. This enables us to integrate two ranking perspectives: F solely considers the error propagation across failing traces, and $F \cup P$ partially considers coverage of passing traces besides the propagation of errors. We will show in our experimental evaluation that this ranking aggregation achieves better performance than either approach alone.

Trace compression. Many test cases in the Defects4J suite execute long running loops, which increases the overhead of EPG construction and slows down marginal inference. On the other hand, these long running loops produce regular and repetitive data and control dependencies. This allows us to eliminate repeated loop iterations to keep the EPG compact, and accelerate both its construction and subsequent Bayesian inference.

Given a trace $\tau = t_1, t_2, \ldots t_n$, a loop is identified by a subsequence of statements $t_i, t_{i+1}, \ldots, t_{i+\ell}$ repeated consecutively in τ. To preserve data and control dependencies within a loop, it suffices to retain two iterations to capture dependencies both upon *entering* and *exiting* the loop. In this case, data and control dependencies from the remaining iterations are identical to the last iteration. We use the instrumentation-guided loop identification algorithm shown in Algorithm 1 to find and compress loop iterations. Note that ISLOOP is a decision algorithm with time complexity $O(n)$, straightforward to implement using dynamic programming and KMP.

Algorithm 1: Compress(τ, γ), where $\tau = \langle t_1, t_2, \ldots, t_n \rangle$ is a trace, and $\gamma > 1$ is the minimum desired repetition period.

1. For each program statement l that is a loop entry point, define:
$$\chi(l) = [i \mid \text{Stmt}(t_i) = l].$$
2. Over all loop entry points l, in decreasing order of number of occurrences, $|\chi(l)|$:
   A. Let $\chi(l) = [i_1, i_2, \ldots, i_k]$.
   B. For each $i_a \in \chi(l)$, if IsLoop$(\tau_{i_a}, \ldots, \tau_{i_{a+\gamma}})$:
      a. Find the largest $b \geq a + \gamma$ such that IsLoop$(\tau_{i_a}, \ldots, \tau_{i_b})$. ▷ Can be done through Binary Search
      b. Let L be the loop period reported by IsLoop.
      c. Let $\sigma = t_{i_b-2L}, \ldots, t_{i_b-1}$ and $\tau' = t_1, t_2, \ldots, t_{i_a-1}, \sigma, t_{i_b}, \ldots, i_n$. ▷ Keep 2 iterations of the loop
      d. Return Compress(τ', γ).
3. If $\gamma > 3$, return Compress$(\tau, \gamma - 1)$. ▷ No more loops with minimum period γ could be found
4. Return τ. No more loops could be found.

When instrumenting programs at the IR level in Soot, we identify `goto` statements l_g along with their corresponding target statements l_t, and selectively insert annotations immediately before l_t if l_t.`lineNumber` $\leq l_g$.`lineNumber`. This represents a jump within a procedure to one of its preceding statements, which naturally represents a backedge in the CFG. We use this instrumentation to identify statements that serve as loop entry points in a given trace.

Aliasing. Another challenge arises when multiple variables point to the same memory location and defining a variable u might result in an "*action-at-a-distance*" update to another variable v. A thorough solution to this problem would require careful modeling of the heap. We instead perform a lightweight analysis to create the set Ref of all assignment statements whose right-hand side is a reference variable. Next, when computing Def(l), we consult Ref to determine if there is an aliasing variable w for each $v \in$ Def(l). In this case, we add w to Def(l) and transitively perform this augmentation until fixpoint. Note that we assume an aliasing remains persistent through the execution, which is guaranteed by the SSA-like format that Soot provides.

Pruning dependencies. Finally, recall that the conditional probability distributions defined in Equation 2 contain 2^k entries, where $k = |\text{Pred}(v)|$ is the in-degree of the vertex v. In particular, this exponential growth of the CPD tables causes challenges with: (*a*) invocation statements with several arguments, and with (*b*) virtual variables `cfx` that may depend on many vertices.

To avoid space explosion in (*a*), in the case of many-argument functions (i.e., where $|\text{FArgs}(t_i)| > 10$), we delete edges from argument definitions to $v \in V_i$. This optimization may be alternatively characterized as disabling Rule R_{DU} for many-argument invocation statements. Note that these dependencies can still be recovered by transitively following inter-procedural edges, so the optimization does not lead to a total loss of error propagation pathways.

To address challenge (*b*), arising from `cfx` variables with high in-degree, we break dependencies with introducing new variables $\texttt{cfx}_1, \ldots, \texttt{cfx}_n$, each of which is dependent on at most 10 other vertices and which are themselves connected to the main virtual variable `cfx`. This limits the size of the CPDs to 2^{10} entries, while leading to at most a linear increase in the number of EPG vertices.

5 Experimental Evaluation

Our evaluation seeks to answer the following questions:

RQ1. How effective is Prosecutor in localizing faults?
RQ2. How long does Prosecutor take to compute its ranking?

RQ3. How do passing and failing traces affect ranking quality?
RQ4. How do counterfactual executions affect the accuracy?

Experimental setup. We conducted all our experiments on a workstation machine with an AMD Ryzen 9 5950X CPU and 128 GB of memory running Ubuntu 22.04. We set a timeout of 20 hours per faulty version for Prosecutor and the baselines.

Benchmarks. We evaluate our approach on 470 faulty versions of 13 projects from the Defects4J (v3.0.1) benchmark suite [11]. These faulty versions contain, on average, 21K lines of source code. In addition, each failing test on average consists of a trace of length 22,000 steps and executes ≈400 distinct lines of code overall. Even though loop compression and dependency pruning massively improve the overall scalability of our approach, the current implementation still faces challenges when analyzing recursive programs or those with hundreds of sparse loops (as we need to keep 2 iterations of each loop). In such cases, both EPG construction and the subsequent marginal inference process incur substantial computational overhead. Among the total 542 versions of these 13 projects, we thus exclude benchmarks whose failing traces are longer than 500,000 Jimple statements—there are 61 such buggy versions in all. We also exclude 11 benchmarks with incorrect function declarations or missing method bodies, where no *executable* line can be blamed.

Baselines. We compare Prosecutor to 8 existing techniques including:

(1) four SBFL approaches: Op2 [22], Tarantula [10], DStar [29], and Ochiai [3],
(2) two MBFL approaches: MUSE [21] and Metallaxis [23], and
(3) two other state-of-the-art methods: SmartFL [35] and DepGraph [24], respectively.

We reimplemenented SBFL and MBFL approaches using Soot analysis framework [27] to get coverage profiles through instrumentation and by using Major [2] to generate tentative mutants. We also used the implementations of SmartFL and DepGraph provided in their available artifacts.

5.1 RQ1: Effectiveness of Ranking

We started by identifying the actual faulty lines in all benchmarks. For 4 projects, (i.e., Lang, Math, Time, Chart), we reused the labeling provided by SmartFL, and for the remaining projects we consulted the patches provided by Defects4J to derive the expected ground truth. Next, we ran both our system and the baselines on all benchmarks and noted the position of the *faulty line* in each proposed ranking. We aggregate these results in Table 3, using (*a*) top-*k* measure that shows the number of project versions whose bug was successfully identified within the top *k* entries of the reported rankings, and (*b*) the median EXAM score, which measures the median percentage of code examined to locate the faulty statement in each ranking.

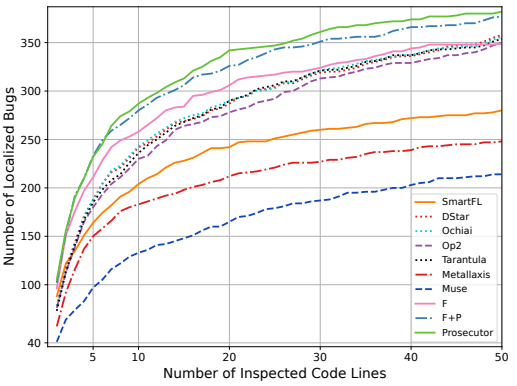

Fig. 9. Number of faults in top-*k* predictions of each technique across the entire dataset. The ideal technique would place the true fault location as its first prediction, so faster rising curves indicate higher effectiveness.

Overall, we observe that Prosecutor allows for 40% of true fault locations to be identified by examining just 3 lines of code. Alternatively, the median EXAM score reveals that half of all faulty locations can be identified by examining just 4.5% of the statements covered by failing tests. We include an alternative visual presentation of these

Table 3. Effectiveness of Prosecutor and the baselines in statement-level fault localization. The best performing approach in each case is marked in bold. The column labeled "w/o Cfx" indicates the results without counterfactual analysis, and the columns labeled F and F ∪ P show the results of two ablated settings without ranking aggregation.

Project	Metric	DStar	Ochiai	Op2	Tarantula	Muse	Metallaxis	SmartFL	Prosecutor	w/o Cfx	F	F ∪ P
Lang (58)	TOP-1	7	7	7	8	8	13	**20**	**20**	21	19	20
	TOP-3	20	20	20	21	16	26	30	**31**	28	31	32
	TOP-5	25	26	23	26	22	34	34	**41**	38	36	41
	TOP-10	39	38	39	38	29	37	41	**49**	48	48	52
	TOP-20	45	42	45	43	35	45	50	**56**	54	54	55
	EXAM	14.0	14.1	14.1	14.5	21.2	9.4	7.0	**6.0**	6.3	6.7	5.1
Compress (45)	TOP-1	10	9	10	8	2	9	10	**13**	13	13	12
	TOP-3	14	13	14	12	9	17	15	**19**	18	19	18
	TOP-5	19	17	19	16	12	18	16	**21**	20	22	21
	TOP-10	22	21	22	21	20	22	19	**28**	27	29	24
	TOP-20	23	22	23	23	22	27	23	**34**	34	33	27
	EXAM	14.4	14.8	14.4	14.4	20.0	8.3	16.3	**5.0**	5.0	3.9	7.3
Chart (26)	TOP-1	3	3	2	2	2	2	**7**	5	5	5	5
	TOP-3	7	7	6	7	3	5	12	**17**	17	14	18
	TOP-5	10	10	9	10	4	6	17	**21**	21	18	21
	TOP-10	11	12	10	12	7	8	19	**23**	22	20	24
	TOP-20	14	16	15	16	8	10	21	**25**	24	23	25
	EXAM	8.3	7.6	9.0	8.3	35.4	33.8	3.7	**2.0**	1.5	2.9	2.0
JackCore (25)	TOP-1	4	4	5	4	1	4	**6**	**6**	6	5	5
	TOP-3	8	7	8	8	3	6	7	**13**	12	12	12
	TOP-5	10	10	10	10	4	11	9	**14**	12	12	15
	TOP-10	12	11	12	12	6	12	11	**17**	15	15	17
	TOP-20	13	14	13	15	8	12	15	**18**	16	16	18
	EXAM	4.0	5.2	3.9	3.5	26.6	3.9	4.3	**2.0**	2.8	3.7	2.0
Math (82)	TOP-1	15	15	14	14	12	9	**15**	13	13	12	13
	TOP-3	25	25	23	25	17	20	25	**28**	27	27	28
	TOP-5	29	29	27	30	22	27	30	**38**	36	37	36
	TOP-10	35	37	33	37	31	35	40	**49**	48	45	44
	TOP-20	43	45	41	45	41	40	43	**56**	54	56	54
	EXAM	9.1	9.1	10.3	8.3	16.9	10.3	9.0	**5.2**	5.9	5.0	6.5
Jsoup (60)	TOP-1	6	6	5	6	3	4	9	**14**	13	14	15
	TOP-3	12	12	11	13	4	11	11	**20**	20	18	20
	TOP-5	18	17	18	16	6	14	12	**24**	24	22	25
	TOP-10	27	27	25	22	7	20	18	**28**	29	28	28
	TOP-20	33	**34**	31	31	10	22	20	**34**	34	32	30
	EXAM	**2.9**	**2.9**	3.2	3.4	100	100	18.2	**2.9**	2.9	4.6	3.9
Cli (39)	TOP-1	**9**	8	**9**	8	6	4	6	7	7	6	7
	TOP-3	**13**	12	12	11	7	10	9	10	9	11	10
	TOP-5	**18**	17	17	17	8	13	13	11	10	12	11
	TOP-10	**21**	**21**	20	20	9	17	14	18	17	15	16
	TOP-20	**24**	**24**	23	**24**	12	19	18	23	21	20	23
	EXAM	**8.3**	**8.3**	10.7	**8.3**	32.3	12.2	16.3	12.5	13.1	13.2	9.2

Project	Metric	DStar	Ochiai	Op2	Tarantula	Muse	Metallaxis	SmartFL	Prosecutor	w/o Cfx	F	F ∪ P
Codec (16)	TOP-1	3	3	3	2	2	3	1	**5**	4	5	5
	TOP-3	5	5	5	5	4	3	2	**6**	6	6	5
	TOP-5	5	5	5	5	4	5	3	**6**	6	8	6
	TOP-10	6	6	6	6	6	8	3	**9**	9	9	7
	TOP-20	11	11	**13**	11	8	9	3	12	12	12	12
	EXAM	15.1	15.1	13.9	15.1	20.3	14.4	100	**9.2**	9.2	7.2	12
Mockito (38)	TOP-1	6	7	6	6	1	2	2	**7**	7	6	7
	TOP-3	12	13	11	10	3	5	3	**14**	14	13	14
	TOP-5	16	**18**	15	16	5	5	3	14	14	13	16
	TOP-10	19	**20**	17	19	5	7	3	18	18	15	19
	TOP-20	22	22	19	22	6	7	3	**25**	25	22	23
	EXAM	2.3	2.0	2.6	2.3	100	100	100	**1.4**	1.4	2.4	1.3
Time (26)	TOP-1	2	2	2	2	1	2	2	**3**	i	j	k
	TOP-3	**5**	4	**5**	**5**	2	5	3	**5**	i	j	k
	TOP-5	**10**	9	9	9	3	7	5	8	i	j	k
	TOP-10	**12**	**12**	11	11	4	7	10	**12**	i	j	k
	TOP-20	15	15	15	**16**	5	10	14	15	i	j	k
	EXAM	1.5	1.5	1.5	**0.9**	30.0	6.3	1.3	1.2	1.2	0.9	1
Csv (16)	TOP-1	4	4	4	3	2	3	4	**6**	6	6	5
	TOP-3	6	6	6	5	3	5	7	**7**	7	7	6
	TOP-5	**7**	**7**	**7**	**7**	4	**7**	**7**	**7**	7	7	6
	TOP-10	**8**	**8**	**8**	**8**	6	7	7	7	7	7	7
	TOP-20	**12**	**12**	**12**	**12**	7	8	8	8	8	7	7
	EXAM	**8.5**	**8.5**	**8.5**	**8.5**	32.5	24.0	13.9	19.4	16.3	28.9	19.9
Gson (17)	TOP-1	4	**5**	4	**5**	1	2	3	3	3	3	3
	TOP-3	7	7	7	**8**	3	3	4	7	8	6	8
	TOP-5	7	7	7	8	3	3	4	**11**	10	8	10
	TOP-10	**13**	11	10	11	3	3	5	11	10	9	10
	TOP-20	**14**	**14**	11	13	3	3	5	13	13	11	15
	EXAM	2.5	2.5	5.0	3.4	100	100	100	**1.9**	2.2	5.0	2.0
Collections (22)	TOP-1	**5**	**5**	**5**	**5**	_	_	3	3	3	2	3
	TOP-3	10	11	11	11	_	_	7	**13**	14	12	14
	TOP-5	14	15	14	15	_	_	11	**16**	16	16	17
	TOP-10	17	**19**	17	**19**	_	_	14	18	18	18	19
	TOP-20	18	19	17	19	_	_	19	**21**	21	20	21
	EXAM	7.0	5.8	6.2	5.8	_	_	9.7	**5.6**	5.6	5.8	4.9
Total (470)	TOP-1	78	78	76	73	41	57	88	**105**	104	96	102
	TOP-3	144	142	139	141	74	116	135	**190**	186	176	192
	TOP-5	188	187	180	185	97	150	164	**232**	222	211	233
	TOP-10	242	243	230	236	133	183	204	**287**	279	258	280
	TOP-20	287	290	278	290	165	212	242	**342**	331	306	326
	EXAM	7.0	7.1	7.6	7.0	26.5	17.8	12.0	**4.5**	4.5	4.8	4.6

results for the entire dataset in Figure 9. Observe that the technique significantly outperforms *all* of the baselines, and identifies 32% more bugs within its top 3 predictions than the best-performing one. Figure 9 also demonstrates that to identify fault locations in 50% of the benchmarks, a developer using Prosecutor would need to inspect 44% fewer lines than even the best-performing baseline.

Our technique demonstrates dramatic improvements on bugs associated with the projects such as Compress and Jsoup. While it is hard to pinpoint an exact cause, we speculate that this is because of challenges faced by SBFL and MBFL techniques when the faulty line is frequently executed by both passing and failing tests.

Notice that SBFL approaches occasionally outperform Prosecutor (notably in the Cli project). We believe that this is due to a combination of factors, including unexpectedly deep faults and imperfect modeling in the EPG (such as the lack of aliasing information). Addressing these limitations is a good direction of future work.

Note 5.1. Unfortunately, we could not run MBFL on the Collections project as it needs Java versions not supported by Major.

Table 4. Comparison of PROSECUTOR and DepGraph in method-level fault localization. The best performing approach in each case is marked in bold.

Project	Method	TOP-1	TOP-2	TOP-3	TOP-4	TOP-5
Lang (58)	DepGraph	42	47	51	54	55
	PROSECUTOR	**44**	**55**	**56**	**56**	**56**
Compress (45)	DepGraph	23	30	32	35	37
	PROSECUTOR	**29**	**36**	**39**	**41**	**41**
Chart (26)	DepGraph	**15**	16	20	21	21
	PROSECUTOR	12	**21**	**24**	**24**	**25**
JackCore (25)	DepGraph	**10**	13	14	14	14
	PROSECUTOR	8	**14**	**17**	**18**	**20**
Math (82)	DepGraph	**51**	**64**	**69**	**75**	**75**
	PROSECUTOR	43	55	61	67	72
Jsoup (60)	DepGraph	17	25	28	29	31
	PROSECUTOR	**20**	**25**	**32**	**32**	**33**
Mockito (38)	DepGraph	**17**	**23**	**26**	**29**	**30**
	PROSECUTOR	**17**	22	25	28	28
Codec (16)	DepGraph	7	9	10	10	11
	PROSECUTOR	**8**	**10**	**11**	**14**	**14**
Time (26)	DepGraph	**14**	15	**16**	**16**	17
	PROSECUTOR	10	**16**	**16**	**16**	**18**
Cli (39)	DepGraph	**16**	**21**	**22**	**27**	**28**
	PROSECUTOR	10	15	17	21	23
Csv (16)	DepGraph	5	7	7	9	10
	PROSECUTOR	**8**	**9**	**10**	**10**	**11**
Gson (17)	DepGraph	**13**	**13**	**14**	**14**	**15**
	PROSECUTOR	5	**13**	**14**	**14**	14
Collections (22)	DepGraph	–	–	–	–	–
	PROSECUTOR	6	14	17	18	20
Total (470)	DepGraph	**230**	283	309	333	344
	PROSECUTOR	220	**305**	**339**	**359**	**376**

Comparison to learning-based techniques. We separately present the comparison of PROSECUTOR to DepGraph, a recent state-of-the-art learning based fault localization technique, in Table 4. We separate these results because DepGraph only performs *method-level* localization. In order to perform an apples-to-apples comparison with PROSECUTOR, we coarsen our results by choosing the most suspicious statement within each method as a representative for the method as a whole.

Apart from two projects, Math and Mockito, PROSECUTOR consistently matches or surpasses DepGraph in its top 2, 3, 4, and 5 predictions for the most suspicious methods. Our approach even outperforms DepGraph at the first prediction in several projects, including Lang and Compress, albeit in general DepGraph demonstrates superior prediction for the top 1.

The superior top-1 performance of DepGraph can be reasonably explained by its design focus on method-level fault localization, in contrast to the statement-level suspiciousness scores that are reported by PROSECUTOR. Furthermore, their technique relies on additional sources of information such as the history of code changes, which our approach does not take into account.

Note 5.2. (*a*) We were unable to run DepGraph on the Collections project because of compatibility issues with the most recent version of Defects4J. (*b*) In contrast to our implementation, DepGraph does not consider the possibility of the bug being present in the initialization portion of the test suite. In order to keep the comparison uniform, our statistics for PROSECUTOR in Table 4 similarly exclude these methods from consideration.

5.2 RQ2: Runtime Overhead of PROSECUTOR

We also measured the total time that each approach needs to provide its ranking of possible fault locations. PROSECUTOR requires an average of 240 seconds to localize each fault (using geometric means). EPG construction (56%), marginal inference (15%) and counterfactual analysis (5%) turn out to be the most significant contributors to this running time. Figure 10 shows the breakdown of these statistics across different projects.

Unsurprisingly, we observed that SBFL techniques are consistently the most lightweight, and MBFL approaches are the most expensive, requiring on average 75 and 770 seconds per benchmark, respectively. We also observed that SmartFL is ≈2× faster than our approach. Most of this gap in running time is due to less accurate and faster graph construction, for which SmartFL provides a parallelized implementation. In principle, we can also parallelize EPG construction, which we expect to lead to similar speedups.

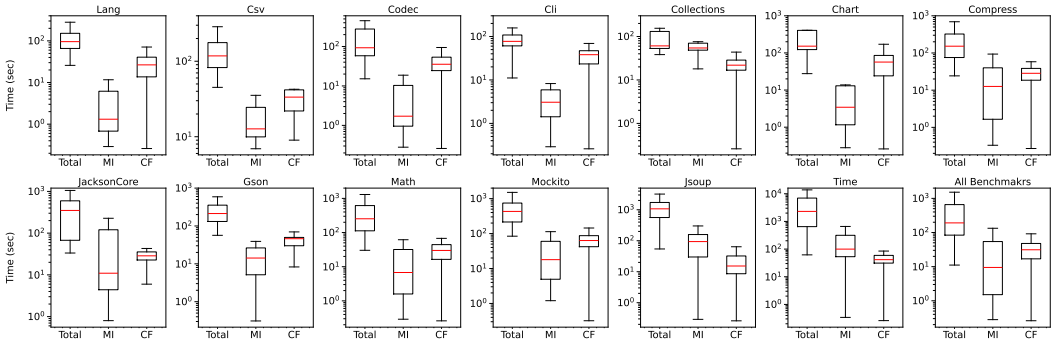

Fig. 10. PROSECUTOR running time breakdown, including the total time and the time needed for performing for marginal inference (MI) and counterfactual analysis (CF).

Our implementation for PROSECUTOR provides an average graph construction throughput of 70 stmt/sec, and the loop identification algorithm could reduce the length of traces with an average speed of 710 stmt/sec. Furthermore, our experiments show that the loop reduction on average eliminated more than 38,000 statements from the processed traces in each benchmark. This clearly highlights the contribution of loop identification algorithm in the overall scalability.

5.3 RQ3: Effect of Passing and Failing Traces

We also ran PROSECUTOR in two ablated settings to measure the impact of failing and passing traces on the eventual ranking quality. We thus eliminate the ranking aggregation phase and individually consider the two rankings obtained by constructing EPG from failing traces, F, and by using both failing and passing traces, $F \cup P$.

We aggregated these statistics using the top-k measure in both Table 3 and Figure 9. As shown in Figure 9, even the first setting, which does not leverage passing traces, outperforms the best-performing baseline in the top 20 predictions. Note that in Defects4J, on average less than 2 failing tests trigger the bug for each benchmark. Thus, one notable advantage of our approach is its ability to effectively localize the fault even without access to passing traces by using a few failing tests. This is essentially improbable when using SBFL techniques.

On the other hand, by including the dependency information from passing traces in the EPG, PROSECUTOR can significantly improve the quality of its ranking. Overall, PROSECUTOR in the second setting ($F \cup P$) could identify 10% and 9% more true faults in its top 5 and top 10 predictions, respectively, compared to the first setting.

5.4 RQ4: Role of Counterfactual Executions

To assess the impact of counterfactual analysis on the overall effectiveness of the rankings, we performed a final experiment with the feature turned off. Table 3 also includes the results of this study. In most cases, observe the number of faults identified within the top-k entries of the ranking slightly drops without counterfactual observations (see w/o Cfx column).

Overall, results show that counterfactual analysis increased the number of faults identified in top 5 predictions by 5%. It also boosts the number of localized bugs in top 10 and 20 predictions consistently by 3%.

We specifically observed that counterfactual analysis helps in raising the priority of deep bugs in the ranking. From a high-level view, program elements that are far away in the EPG from the failure point typically appear less suspicious. On the other hand, marking the virtual variable `cfx`

as incorrect draws suspicion towards the statement nodes with edges leading to `cfx` vertex and their surrounding statements.

For example, version #17 of the Cli project in Defects4J has a bug that is 24 edges away from assertion violation in the EPG. This causes the actual fault to be placed in rank 52 in the ablated setting, while the full Prosecutor places this statement at the 16th position of its ranking. This evidently shows the effect of counterfactual analysis in prioritizing deep bugs.

A broader view of this improvement across all benchmarks with deep bugs is illustrated in Figure 11. We define bug depth as the number of edges in the EPG from the buggy statement to the failure location. We consider bugs whose depths exceed the third quartile (i.e., $\geq$ 13 edges) as "*deep* bugs". Compared to SmartFL, Prosecutor without counterfactual analysis can localize 60% and 69% more deep faults within its top 10 and 20 predictions, respectively. Incorporating counterfactual analysis into the method further enhances the accuracy of localization and allows our approach to identify 100% and 94% more deep bugs than SmartFL within its top 10 and 20 predictions, respectively.

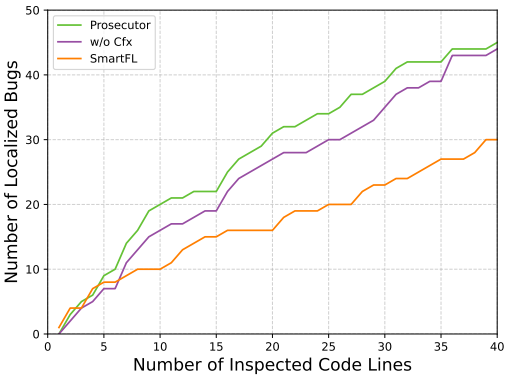

Fig. 11. Number of deep faults in top-k predictions of Prosecutor–with and without counterfactual analysis–against SmartFL. The benchmarks totally contain 111 deep faults. The faster rising curves are better.

We also observed that in 38% of the benchmarks, at least one of our counterfactual experiments change the eventual test outcome. This clearly shows that the initial Bayesian inference can effectively prioritize suspicious branch conditions seemingly related to the fault, which again highlights the effectiveness and precision of our modeling.

6 Limitations of Our Approach

We now briefly discuss some limitations of Prosecutor that suggest directions for future research.

First, our approach is targeted towards "*errors of commission*", e.g., errors involving incorrect values, assertion violations, program crashes, and unexpected exceptions. In contrast, some bugs manifest as "*errors of omission*", mainly in the form of expected exceptions that not actually raised. We handle such cases by performing a lightweight analysis to identify and annotate possible failure points where the exception might arise. A general solution to this problem is a good direction for future work.

Next, recall that the EPG provides a model for how errors arise during program executions. Although the EPG successfully captures a large number of errors occurring within the Defects4J programs, it is unable to model complex patterns of aliasing (our current solution uses static specifications to identify possibly aliasing statements) or adequately model the heap. More precise modeling of data dependencies would presumably improve the overall effectiveness.

Finally, our current implementation of Prosecutor faces difficulties in scaling to traces with millions of statements. In the Defects4J dataset, this mostly arises from programs with repeated irregular invocations of the same iterative computation, which limits the effectiveness of the trace compression procedure. Note that the EPG construction algorithm requires $O(n)$ time and makes a single pass over the execution trace with n statements. The main performance bottleneck in our implementation lies in the I/O overhead of exchanging data between Java and Python processes. Besides parallelization of the graph construction, performing static dependency analysis beforehand

in an offline process can significantly reduce I/O overhead and further enhance the scalability. Another interesting direction of future work is to identify and prune unnecessary dependencies in the EPG to make the fault localization both faster and more effective.

7 Related Work

There is a large body of research on fault localization and probabilistic program reasoning. We now summarize the most closely related work. For a detailed survey, we refer readers to [30].

Fault localization. The introduction of delta debugging for simplifying and isolating failure-inducing inputs [34] was perhaps among the earliest work that sparked a line of research towards both automated debugging and fault localization techniques. Research on fault localization began with spectrum-based approaches, which rank program statements by comparing their execution frequencies in passing and failing tests [3, 4, 10, 22, 29]. It was later developed by mutation-based techniques, which inspect test behaviors across a set of mutants to assess the impact of changing each statement on test outcomes [7, 21, 23]. Predicate switching [36] alters control flow by heuristically identifying critical predicates and inverting conditional outcomes during failing runs, which may be regarded as a simple form of counterfactual analysis. In contrast, our work describes how to effectively incorporate the results of these experiments into a single unified probabilistic model. History-based techniques prioritize fault candidates based on historical defect data [12, 26] instead of relying on information from test runs. This data may be combined with coverage profiles to make the fault identification more accurate [28].

Besides statistical approaches, learning-based fault localization has recently gained significant attention. UniVal [15] uses statistical causal inference to estimate the average causal effect of counterfactual assignments to program variables, without actually executing the program. Neural-MBFL [9] uses a pre-trained model that can leverage the context information surrounding the mutation position to improve the quality of mutants generated. GMBFL [31] models the relationship between code entities, mutants, and test runs in a graph representation and utilizes gated graph attention neural networks to learn useful features from this graph.

Another line of work on learning-based method-level fault localization was started by DeepFL [17], which was further enhanced by newer classifiers. DeepFL incorporates suspiciousness scores of SBFL and MBFL with code complexity and text similarity into a unified deep-learning model. Grace [18] leverages graph-based representation learning to fully utilize coverage information that involves connective relationships between tests and program entities. DepGraph [24] proposes a more compact representation by integrating the inter-procedural call graph into the GNN model alongside code change information as an additional feature. HetFL [5] mitigates data imbalance and inefficient representation of previous methods by resampling faulty data and modeling programs as heterogeneous graphs. To our knowledge, most of these techniques perform method-level localization. This limits their effectiveness when applied to failing unit tests, which typically only execute a small number of functions (In our experiments, failing tests in Defects4j dataset executed a median of 19 and an average of 30 functions before crashing). In addition, the non-reliance on training data and the non-necessity of a GPU makes our technique much more broadly applicable and less expensive than most learning-based approaches.

Probabilistic program reasoning. Starting with Eugene [19] and Bingo [25], there is a growing body of work on using Bayesian inference to probabilistically reason about programs. Xu et al. model debugging as a probabilistic inference problem, where human-like reasoning rules are modeled as conditional probability distributions [32]. Inspired by this work, SmartFL [35] constructs an efficient probabilistic model of program semantics using dynamic data and static control dependencies, which presents a promising direction, but the proposed encoding of program semantics is subject

to several limitations: SmartFL cannot sufficiently model executions where errors are propagated across procedures through complicated inter-procedural control and data dependencies. It also does not consider (*a*) control dependencies introduced by non-conventional control flow constructs such as exceptions, and (*b*) the impact of branch conditions on values in the case of unexplored branches. SmartFL solely relies on observations from failed/passed assertions, yet many tests in Defects4J and real-world projects fail for other reasons, e.g., program crashes and unexpected exceptions, without triggering assertion violations.

To address these limitations, we use a fundamentally different approach to model error propagation in the level of source code using EPG with three types of nodes. Our novel inter-procedural analysis enables precise modeling of error propagation across procedures, and the EPG enables locating faults regardless of failure reasons. Unlike SmartFL in which a set of statements are manually identified to be more fault-tolerant, our method of converting EPG into a Bayesian networks considers equal degree of fault-tolerance for all statements, and instead density functions is set up so that the accumulation of errors results in the reduction of the correctness probability. Finally, we leverage a novel way of selecting a set of counterfactual experiments and incorporating observations from them to more accurately reason about fault locations.

8 Conclusion

In this paper, we presented Prosecutor, a new fault localization technique that combines Bayesian reasoning about erroneous traces with a novel counterfactual analysis to effectively identify faults at the statement level. A comprehensive evaluation on benchmarks from the Defects4J dataset of faulty Java libraries indicates that our technique is applicable to real-world programs and sets up a new state-of-the-art in ranking effectiveness.

Data Availability

We have submitted our replication package as supplementary material for the review process. We will also submit the artifact for artifact evaluation and make it publicly available upon paper acceptance.

References

[1] [n. d.]. *Apache Commons Lang java library.* https://commons.apache.org/proper/commons-lang

[2] [n. d.]. *The Major Mutation Framework.* https://mutation-testing.org

[3] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan van Gemund. 2009. A Practical Evaluation of Spectrum-Based Fault Localization. *Journal of Systems and Software* 82, 11 (Nov. 2009), 1780–1792. doi:10.1016/j.jss.2009.06.035

[4] Rui Abreu, Peter Zoeteweij, and Arjan van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 88–99. doi:10.1109/ASE.2009.25

[5] Xin Chen, Tian Sun, Dongling Zhuang, Dongjin Yu, He Jiang, Zhide Zhou, and Sicheng Li. 2024. HetFL: Heterogeneous Graph-Based Software Fault Localization. *IEEE Transactions on Software Engineering* 50, 11 (2024), 2884–2905. doi:10.1109/TSE.2024.3454605

[6] Jong-Deok Choi and Jeanne Ferrante. 1994. Static Slicing in the Presence of Goto Statements. *ACM Transactions on Programming Languages and Systems* 16, 4 (July 1994), 1097–1113. doi:10.1145/183432.183438

[7] Zhanqi Cui, Minghua Jia, Xiang Chen, Liwei Zheng, and Xiulei Liu. 2020. Improving Software Fault Localization by Combining Spectrum and Mutation. *IEEE Access* 8 (2020), 172296–172307. doi:10.1109/ACCESS.2020.3025460

[8] Ron Cytron, Jeanne Ferrante, Barry Rosen, Mark Wegman, and Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (Oct. 1991), 451–490. doi:10.1145/115372.115320

[9] Bin Du, Baolong Han, Hengyuan Liu, Zexing Chang, Yong Liu, and Xiang Chen. 2024. Neural-MBFL: Improving Mutation-Based Fault Localization by Neural Mutation. In *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. 1274–1283. doi:10.1109/COMPSAC61105.2024.00168

[10] James Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE).*

ACM, 273–282. doi:10.1145/1101908.1101949

[11] René Just, Darious Jalali, and Michael Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 437–440. doi:10.1145/2610384.2628055

[12] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. Predicting Faults from Cached History. In *29th International Conference on Software Engineering (ICSE)*. 489–498. doi:10.1109/ICSE.2007.66

[13] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press.

[14] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques - Adaptive Computation and Machine Learning*. The MIT Press.

[15] Yiğit Küçük, Tim Henderson, and Andy Podgurski. 2021. Improving Fault Localization by Integrating Value and Predicate Based Causal Inference Techniques. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. IEEE Press, 649–660. doi:10.1109/ICSE43902.2021.00066

[16] Tom Leonard, John S. J. Hsu, and Kam-Wah Tsui. 1989. Bayesian Marginal Inference. *J. Amer. Statist. Assoc.* 84, 408 (1989), 1051–1058. arXiv:https://www.tandfonline.com/doi/pdf/10.1080/01621459.1989.10478871 doi:10.1080/01621459.1989.10478871

[17] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 169–180. doi:10.1145/3293882.3330574

[18] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 664–676. doi:10.1145/3468264.3468580

[19] Ravi Mangal, Xin Zhang, Aditya Nori, and Mayur Naik. 2015. A User-Guided Approach to Program Analysis. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 462–473.

[20] Joris Mooij. 2010. libDAI: A Free and Open Source C++ Library for Discrete Approximate Inference in Graphical Models. *Journal of Machine Learning Research* 11, 74 (2010), 2169–2173. http://jmlr.org/papers/v11/mooij10a.html

[21] Seokhyeon Moon, Yunho Kim, Moonzo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *IEEE Seventh International Conference on Software Testing, Verification and Validation*. 153–162. doi:10.1109/ICST.2014.28

[22] Lee Naish, Hua Jia Lee, and Kotagiri Ramamohanarao. 2011. A Model for Spectra-Based Software Diagnosis. *ACM Transactions on Software Engineering and Methodology* 20, 3, Article 11 (Aug. 2011), 32 pages. doi:10.1145/2000791.2000795

[23] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-Based Fault Localization. *Journal of Software: Testing, Verification and Reliability* 25, 5–7 (Aug. 2015), 605–628. doi:10.1002/stvr.1509

[24] Md Nakhla Rafi, Dong Jae Kim, An Ran Chen, Tse-Hsun (Peter) Chen, and Shaowei Wang. 2024. Towards Better Graph Neural Network-Based Fault Localization through Enhanced Code Representation. *Proc. ACM Softw. Eng.* 1, FSE, Article 86 (July 2024), 23 pages. doi:10.1145/3660793

[25] Mukund Raghothaman, Sulkeha Kulkarni, Kihong Heo, and Mayur Naik. 2018. User-Guided Program Reasoning Using Bayesian Inference, In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation. *SIGPLAN Notices* 53, 4, 722–735. doi:10.1145/3296979.3192417

[26] Jeongju Sohn and Shin Yoo. 2017. FLUCCS: Using Code and Change Metrics to Improve Fault Localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 273–283. doi:10.1145/3092703.3092717

[27] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot: A Java Bytecode Optimization Framework. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM Press, 13.

[28] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2021. Historical Spectrum Based Fault Localization. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2348–2368. doi:10.1109/TSE.2019.2948158

[29] Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Transactions on Reliability* 63, 1 (2014), 290–308. doi:10.1109/TR.2013.2285319

[30] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, Franz Wotawa, and Dongchen Li. 2023. *Software Fault Localization: An Overview of Research, Techniques, and Tools*. John Wiley & Sons, Ltd, Chapter 1, 1–117. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119880929.ch1 doi:10.1002/9781119880929.ch1

[31] Shumei Wu, Zheng Li, Yong Liu, Xiang Chen, and Mingyu Li. 2023. GMBFL: Optimizing Mutation-Based Fault Localization via Graph Representation. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 245–257. doi:10.1109/ICSME58846.2023.00033

[32] Zhaogui Xu, Shiqing Ma, Xiangyu Zhang, Shuofei Zhu, and Baowen Xu. 2018. Debugging with Intelligence via Probabilistic Inference. In *40th IEEE/ACM International Conference on Software Engineering (ICSE)*. 1171–1181. doi:10.1145/3180155.3180237

[33] Xiao Yu, Jin Liu, Zijiang Yang, and Xiao Liu. 2017. The Bayesian Network Based Program Dependence Graph and its Application to Fault Localization. *Journal of Systems and Software* 134 (2017), 44–53. doi:10.1016/j.jss.2017.08.025

[34] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. doi:10.1109/32.988498

[35] Muhan Zeng, Yiqian Wu, Zhentao Ye, Yingfei Xiong, Xin Zhang, and Lu Zhang. 2022. Fault Localization via Efficient Probabilistic Modeling of Program Semantics. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. ACM, 958–969. doi:10.1145/3510003.3510073

[36] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating Faults Through Automated Predicate Switching. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*. ACM, 272–281. doi:10.1145/1134285.1134324