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

Bayesian reasoning has emerged as a promising approach to fault localization, where the introduction of errors and their subsequent propagation through faulty executions is treated as a stochastic process. One can then perform Bayesian inference on a probabilistic model encoding the program execution to associate individual statements and values with a posterior probability of being erroneous. In this paper, we propose a new graph representation that effectively models error propagation through failing program executions. This structure, which we call the Error Propagation Graph (EPG), extends prior probabilistic approaches by incorporating richer inter-procedural relationships and accounting for the influence of unexplored control-flow branches that may affect variable values. We also show how EPGs can be constructed efficiently and compactly, and how this structure enables the selection of a set of counterfactual experiments, each involving artificially flipping a suspicious branch predicate at runtime and observing its downstream effect on the test outcome. The results of these experiments provide additional evidence that can be incorporated into the EPG to confirm or refute the model's initial suspiciousness estimates. We have implemented this technique in a tool named PROSECUTOR and evaluated it on 470 buggy versions of 13 projects from the Defects4J benchmark suite. Our experimental evaluation shows that PROSECUTOR places 40% of the true fault locations within its top-3 predictions. The technique also significantly outperforms a diverse set of baselines by identifying at least 10%, 11%, 15%, and 19% *more* buggy statements than each of the baselines in its top-1, top-3, top-5, and top-10 predictions, respectively.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: fault localization, debugging, Bayesian reasoning

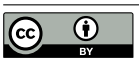
ACM Reference Format:
Sara Baradaran, Yifei Huang, Wei Le, and Mukund Raghothaman. 2026. PROSECUTOR: Bayesian Counterfactual Fault Localization. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 337 (October 2026), 27 pages. https://doi.org/10.1145/3839469

1 Introduction

The fault localization problem can 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 where the bug is most likely to reside. Starting with Jones and Harrold's groundbreaking work on Tarantula [14], 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 monitors test coverage to identify parts of the program that are executed disproportionately more often by failing tests than by passing tests [6, 7, 14, 26, 34], to *mutation-based fault localization (MBFL)*, which applies a series of tentative modifications to the program and observes their downstream effects on

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



ACM 2475-1421/2026/10-ART337
https://doi.org/10.1145/3839469

test outcomes [25, 27]. These approaches have complementary strengths and weaknesses: Spectrum-based techniques are lightweight (they merely 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 bug locations, but because of the large mutation space and the need to repeatedly rerun the entire test suite, their scalability is limited in practice.

Another promising approach to fault localization was recently introduced with SmartFL [37], which constructs a probabilistic model of program execution using data and control dependencies. SmartFL then assigns each statement a prior probability of being faulty and leverages observations from test executions to perform marginal inference on the constructed graph, thereby calculating the posterior probability that each statement is faulty.

Despite demonstrating the effectiveness of probabilistic approaches for fault localization, SmartFL struggles to localize faults in several scenarios. Specifically, the technique has difficulty identifying faults that are located deep along the execution path and whose effects propagate across multiple procedures. Moreover, SmartFL's reliance solely on data and control dependencies is insufficient to fully capture errors arising from incorrect evaluations of branch predicates that prevent expected updates to variables. These limitations highlight the challenges of effectively modeling error propagation in programs with long execution chains and subtle control-flow dependencies.

With this background, our paper extends SmartFL's idea in two ways: *First*, we design a new graph structure, which we call the error propagation graph (EPG), to model the propagation of errors through failing executions more effectively. For example, in addition to direct dependencies between the site of a variable definition and the point of its use, we also include edges in the EPG from unexplored branches that contain unexecuted variable assignments. Such careful modeling contributes to localizing 15% more bugs within the top-5 predictions than SmartFL and to a median EXAM score of 4.4%, less than half that achieved by SmartFL (10.3%).

Next, by attaching conditional probability distributions to each vertex in this graph, we view the EPG as a Bayesian network. We perform Bayesian inference over the constructed network and compute posterior suspiciousness scores for each individual branch predicate evaluated at runtime. We use these posterior probabilities 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 the eventual test outcome. Intuitively, if a branch condition was incorrectly evaluated, then artificially flipping its value may cause the failing test to succeed. Alternatively, if a statement is buggy, then rerouting control flow around that statement may prevent test failure. These experiments therefore help us 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.

Given the availability of the Soot framework [32] and Just et al.'s Defects4J dataset of real-world bugs [15], we implemented our approach for Java fault localization. In doing so, we had to solve several technical challenges: First, marginal inference is famously intractable. We thus need to suitably compress the constructed EPGs to use standard inference algorithms such as loopy belief propagation [18]. Our solution involves an instrumentation-guided loop identification algorithm to detect and compress loops in execution traces. Second, a program execution may evaluate many branch predicates, making exhaustive counterfactual analysis infeasible. We therefore must determine which counterfactual experiments to conduct and how to incorporate their results into the probabilistic model. In particular, we restrict our analysis to the 20 most suspicious branch conditions identified by an initial Bayesian inference run. To leverage information from these experiments, we extend the initial probabilistic model with virtual variables and edges, which allows us to model the impact of flipping a branch condition on its dependent statements as well.

We call our implementation PROSECUTOR and evaluated it on faulty versions of 13 projects from the Defects4J dataset [15]. We compare PROSECUTOR to 8 baselines: 4 from SBFL family, 2 from MBFL family, and 2 state-of-the-art methods, SmartFL [37] and DepGraph [29]. We observed that PROSECUTOR identifies *at least* 15% and 19% more true faulty statements within its top-5 and top-10 predictions than any of the baselines. Moreover, we observed 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.

Contributions. In summary, this paper makes the following contributions:

(1) A systematic approach to model runtime program dependencies in object-oriented programs using a data structure called the error propagation graph (EPG).
(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 redundant dependency elimination, which enable scaling the technique to real-world software.
(4) An empirical evaluation on buggy versions of 13 projects from the Defects4J benchmark, which demonstrates that PROSECUTOR outperforms existing fault localization techniques.

2 Motivating Example

Consider the buggy program in Figure 1, with annotations on the right depicting the execution flow of its two test cases. As shown in the assertions on Lines 32 and 34, the goal of this function is to pad the left side of the input string until it reaches the desired length. In the failing execution, 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.

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

Fig. 1. Implementation of the `leftPad` utility function in the Apache Commons Lang library [3] with a bug on Line 9, which should instead be `padChars = padStr.toCharArray()`.

We argue that neither spectrum-based nor mutation-based techniques can satisfactorily localize this bug: As can be seen from the trace annotations, Lines 24–26 are the only lines of code 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, mutation-based 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 statement on Line 9. In addition, MBFL techniques such as

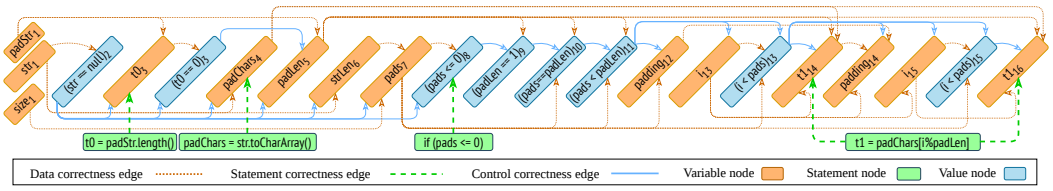


Fig. 2. Snippet of the error propagation graph (EPG) for the trace obtained from `testCase2` in Figure 1.

Metallaxis [27] also consider the effect of the mutation on test *behavior*, rather than simply the eventual *outcome*. In this context, mutating the `if`-statements on Lines 3, 13, 16, 19, and 21 may provide opportunities for premature return, thereby eliminating the crash on Line 26 and instead causing an assertion violation on Line 34. The algorithm therefore characterizes them as "successful" mutations, despite being irrelevant to the bug, which leads to a poor ranking of 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, which are not necessarily induced solely by control and data dependencies.

Also, note that the EPG can be naturally generalized to the setting of having multiple test executions. In these situations, the subgraphs corresponding to different test executions would all share the same set of statement nodes, but would have their own separate sets of value and variable nodes that represent 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. Formally, we write:

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

where $h_0 = \texttt{padStr}_1 \wedge \texttt{(str == null)}_2 \wedge \texttt{(t0 == 0)}_3 \wedge \texttt{padLen = padStr.length()}$ 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 $h_1 = \neg\texttt{padStr}_1 \wedge \texttt{(str == null)}_2 \wedge \texttt{(t0 == 0)}_3 \wedge \texttt{padLen = padStr.length()}$. 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 in the correctness of the vertex, 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 test executions 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

Table 1. The ranked list of suspiciousness scores reported by PROSECUTOR for the 10 most suspicious statements in Figure 1. This is obtained by calculating the posterior probability $\Pr(\neg s \mid \neg \texttt{t1}_{16})$ for each program statement s, given the EPG in Figure 2 as a Bayesian network and the observation $\neg \texttt{t1}_{16}$ on the failing test 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	0.174841	25	`i++`
5	0.172209	12	`pads = size - strLen`
6	0.168113	21	`if (pads < padLen)`
7	0.166703	3	`if (str == null)`
8	0.162269	10	`padLen = padStr.length()`
9	0.156661	25	`i = 0`
10	0.153250	11	`strLen = str.length()`

probability $\Pr(\neg s \mid e)$ that each statement s is buggy, conditioned on our observations e on test behaviors. We show the illustrative results of these calculations in Table 1.

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 is instead located on Line 22. This is a common pitfall: programmers may mistakenly use the wrong method when multiple overloaded methods share the same name. In this scenario, the previously passing test, `testCase1`, now fails as an incorrect string `"L"` (instead of `"!a"`) is returned by the statement `padStr.substring(pads)` on Line 22.

```
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);
29 }
31 public void testCase1() // now an example of failing test
32 { assertEquals("!aabc", StringUtils.leftPad("abc", 5, "!aL")); }        13 1
```

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

Notice that during the execution of `testCase1`, if we surgically flip the value of the branch predicate 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.

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, we perform marginal inference on the constructed graph to calculate the posterior probability $\Pr(\neg c_i \mid e)$, where $e = \neg O\texttt{assert}_{13}$, for each branch predicate c evaluated at Step i and present in the EPG (blue nodes in Figure 2). This procedure creates a ranked list of branch conditions that are suspected of being evaluated incorrectly.

Next, for each condition in the top 20 entries of the ranking, we surgically flip it at runtime and observe its downstream effect on the eventual 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; for example, flipping the branch predicate at Step 11 effectively alters the subsequent test behavior as a result of bypassing the erroneous computation on Line 22.

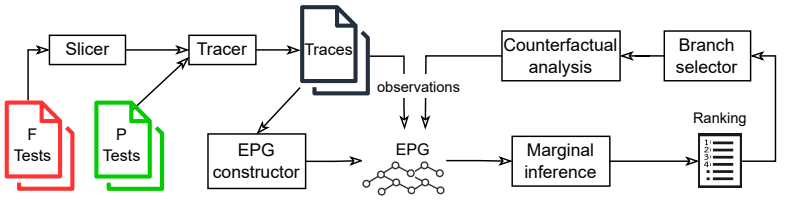


Fig. 4. The Prosecutor fault localization workflow.

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 $\Pr(\neg s \mid e \wedge \neg \texttt{cfx})$ for each statement s as its suspiciousness score. Note that treating `cfx` as an erroneous variable enables the model to consider that any of the nodes with edges leading to `cfx` could be incorrect, and this suspicion will be further propagated to all other nodes from which `cfx` is reachable.

Concretely, feeding the above counterfactual results back into the model will raise the suspiciousness score of Line 22 from 0.47 to 0.61 and place 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 program 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 calculate data dependencies in object-oriented programs, 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, previous research shows that this possibility is not identical across different invocations [9]. 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

Table 2. Defined and used variables in 3-address statements. $\mathcal{R}_a$ denotes reference variables passed as arguments to the callee. Unlike classical Def/Use analysis, we treat both receiver objects and reference arguments as being possibly defined. $O_{\texttt{func}}$ is a dummy variable representing the outcome of the function.

Stmt type	Stmt format	Def	Use
Assign Stmt	t = new Class(a_1, ..., a_n)	$\mathcal{R}_a \cup \{t\}$	$\{a_1, \ldots, a_n\}$
	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)	$\mathcal{R}_a \cup \{t\}$	$\{a_1, \ldots, a_n, u\}$
	u.method(a_1, ..., a_n)	$\mathcal{R}_a \cup \{u\}$	$\{a_1, \ldots, a_n, u\}$
	t = func(a_1, ..., a_n)	$\mathcal{R}_a \cup \{t\}$	$\{a_1, \ldots, a_n\}$
	func(a_1, ..., a_n)	$\mathcal{R}_a \cup \{O_{\texttt{func}}\}$	$\{a_1, \ldots, a_n\}$

designed to perform computations and eventually return the results. We accordingly 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.

To obtain control dependencies, we statically calculate dominance frontiers on the reverse control-flow graph (CFG) of each procedure (RDF relations). This approach determines control dependencies introduced by conventional control-flow constructs (e.g., `if-else`) with explicit edges in CFG [12]. 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 executed instruction $t_i \in \tau$ is an instance of a textual program statement, $s_i = \text{Stmt}(t_i)$.

Vertices. We include all these program statements in the set of *statement* vertices: $V_s = \{s_i \mid t_i \in \tau\}$. Next, we associate every statement $t_i \in \tau$ with a set of vertices V_i. For every variable $v \in \text{Def}(s_i)$ 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 $x \in \text{FArgs}(t_i)$ of its callee function. On the other hand, if t_i does not define any variables, i.e., if $\text{Def}(s_i) = \varnothing$, 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

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

and $V_i^a = \{\langle x@i \rangle \mid x \in \text{FArgs}(t_i)\} \cup \{\langle \texttt{this}@i \rangle\}$, 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

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

$$R_{\text{CTR2}}\ \frac{v \in V_i^d \quad u \in \text{Use}(s_i) \quad \langle u@j \rangle = \text{LastDef}(u, t_i) \quad j < r < i \quad s \in \text{RdfInv}(s_r) \quad u \in \text{Def}(s)}{\langle \#@r \rangle \rightarrow v \in E}$$

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

of callee arguments created by the call statement t_i upon invocation. We collect the set of value and variable vertices: $V_\tau = \bigcup_i V_i$, and define the final set of EPG vertices as:

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

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.

Rule R_s captures the conceptually simplest set of edges: For every executed statement t_i, we include an edge from s_i to each vertex $v \in V_i$. This indicates that the correctness of v depends on the correctness of s_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 $\text{LastDef}(u, t_i)$ 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 $\text{LastCtrl}(t_i)$ 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., s_i is control-dependent on s_j. In this circumstance, Rule R_{CTR1} creates a set of edges from $u \in V_j^d$ to every vertex $v \in V_i^d$. 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 on Line 6, i.e., `if(padStr.length()== 0)`. Nevertheless, if Line 6 was erroneous, then it might have led to 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 s is control-dependent on another statement s' if the outcome of s' determines whether s is executed. Such a definition, however, does not consider the impact of this outcome on the values created at s. To account for the effect of such unexplored branches, we introduce new dependency edges into the EPG using Rule R_{CTR2}.

Let $\text{RdfInv}(s_r)$ be the set of all statements that are statically control-dependent on the statement s_r. Consider an executed 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 s which also defines u. In this case, Rule R_{CTR2} adds an edge from $\langle \#@r \rangle$ to v.

Example 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 `if`-statement on Line 6 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 `if(padStr.length()== 0)` at Line 6, Rule R_{CTR2} derives an edge between the two vertices $\texttt{(t0 == 0)}_3$ and $\texttt{padLen}_5$ in Figure 2.

$$R_{\text{OP1}}\ \frac{\text{Open}(t_i)\quad u \in \text{Use}(s_i)\quad \text{IsArg}(u,t_i)\quad \langle u@j\rangle = \text{LastDef}(u,t_i)\quad x \in \text{FArgsMap}(u,t_i)}{\langle u@j\rangle \rightarrow \langle x@i\rangle \in E}$$

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

$$R_{\text{OP3}}\ \frac{\text{Open}(t_i)\quad u \in \text{Use}(s_i)\quad \text{IsArg}(u,t_i)\quad \langle u@j\rangle = \text{LastDef}(u,t_i)\quad j < r < i\quad s \in \text{RdfInv}(s_r)\quad u \in \text{Def}(s)\quad x \in \text{FArgsMap}(u,t_i)}{\langle \#@r\rangle \rightarrow \langle x@i\rangle \in E}$$

$$R_{\text{OP4}}\ \frac{\text{Open}(t_i)\quad u \in \text{Use}(s_i)\quad \text{IsRcvObj}(u,t_i)\quad \langle u@j\rangle = \text{LastDef}(u,t_i)\quad j < r < i\quad s \in \text{RdfInv}(s_r)\quad u \in \text{Def}(s)}{\langle \#@r\rangle \rightarrow \langle \texttt{this}@i\rangle \in E}$$

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

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, as shown in Figure 6. Although the `throw` statement, inserted by instrumentation, is unreachable at runtime, it creates 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.

```
1  int getSum(int [] arr, int size) {
2    int counter = size, sum = 0;
3    while (true) {
4      counter -= 1;
5      if (counter == 0) // faulty
6        return sum;
7      sum += arr[counter];
8      ...    instrumentation
9    }
10 }
   if (1 == 0) throw an exception;
11 public void failingTest() {
12   int [] arr = {3};
13   int sum = getSum(arr, 1);
14   assertEquals(3, sum);
15 }
```

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 the EPG. Let $\text{Open}(t_i)$ be a predicate denoting whether the execution of the statement t_i in the trace is followed by the start of a new context. Also, let $\text{IsArg}(u,t_i)$ and $\text{IsRcvObj}(u,t_i)$ denote whether the variable u is passed as an argument or the receiver object in statement t_i, respectively. With these definitions, for each operand u, which is last defined by the statement t_j and is passed as the formal argument x of the callee in t_i (i.e., $x \in \text{FArgsMap}(u,t_i)$), Rule R_{OP1} derives an edge from $\langle u@j\rangle$ to $\langle x@i\rangle$. 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. Recall also that $V_i = V_i^d \cup V_i^a$. Therefore, in addition to intra-procedural control dependence edges, Rule R_{CTR1} derives a 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 last statement that controls the execution of the call statement t_i.

Collectively, Rules R_{OP1}–R_{OP4} in Figure 7, together with Rule R_{CTR1} in Figure 5, fully capture inter-procedural control and data dependencies which are required to specify (*a*).

On the other hand, inter-procedural edges that specify (*b*) are derived by Rules R_{CL1}–R_{CL3} when a context is closed. To calculate these edges, we need to first process all the executed statements

$$R_{\text{CL1}}\ \frac{\text{Close}(t_i)\quad t_j = \text{CallStmt}(t_i)\quad v \in V_i^d\quad u \in V_j^d}{v \rightarrow u \in E}$$

$$R_{\text{CL2}}\ \frac{\text{Close}(t_i)\quad t_r = \text{CallStmt}(t_i)\quad v \in \text{Def}(s_r)\quad \text{IsArg}(v, t_r)\quad x \in \text{FArgsMap}(v, t_r)\quad \langle x@j\rangle = \text{LastDef}(x, t_i)\quad r \neq j}{\langle x@j\rangle \rightarrow \langle v@r\rangle \in E}$$

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

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

within the callee context. Let $\text{Close}(t_i)$ be a predicate denoting whether t_i is the last statement executed 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. We also write $\text{CallStmt}(t_i)$ to denote the latest statement t_j executed in the trace before starting the context c_i. These predicates together 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 context closing statement v to all variable vertices u created at the call site.

Finally, let t_r be the call statement. Rule R_{CL2} creates an edge from $\langle x@j\rangle$ to $\langle v@r\rangle$ for each reference variable v that is passed as an argument x, and where t_j is the statement which last defined x in the callee context. Rule R_{CL3} similarly derives a set of edges to the receiver objects.

3.2 A Probabilistic View of the EPG

Conditional probability distributions. Each vertex in the EPG, $G = (V, E)$, represents the correctness of a program element, i.e., either a statement, an evaluated branch predicate, 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 $v \in V$ in 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 concrete 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, i.e., $w = 0$, then $P(v \mid \mathbf{x}_{\text{Pred}(v)}) = 1$. Notice also that the conditional probability of v being correct 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 (i.e., $\text{Pred}(v) = \emptyset$), such as statement vertices, V_s, we uniformly use the prior probability $P(v) = 0.85$.

We note that SmartFL associates each type of statement (arithmetic operations, function calls, etc.) with a manually chosen probability of propagating errors. In contrast, observe that Prosecutor considers all statements equally fault-tolerant, and instead sets up density functions so that the correctness probability of the output is dependent on the number of erroneous inputs.

Bayesian networks. Note that our ultimate goal is to calculate the posterior suspicion $\Pr(\neg s \mid e)$ of each program statement s, given the observations e. Since the EPG is acyclic by construction, the graph together with the CPDs defined above, $P(v \mid \mathbf{x}_{\text{Pred}(v)})$, forms a Bayesian network [20]. We therefore define the following joint probability distribution over the variables in V:

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

where $\mathbf{x}$ is a valuation of all variables in V and x_v denotes the value assigned to variable v under $\mathbf{x}$. It is clearly the case that $\Pr(\mathbf{x}) \geq 0$ and it can be verified that $\sum_{\mathbf{x}} \Pr(\mathbf{x}) = 1$. Hence, $\Pr(\mathbf{x})$ is a well-defined joint 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 all 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 probabilities can be computed by off-the-shelf engines such as LibDAI [24].

3.3 Evidence Collection

To calculate the posterior suspicion of individual elements, we collect evidence about the correctness and incorrectness of runtime values and variables (*a*) from passing and failing executions of the original program; for example, a failed assertion indicates that the enclosed expression evaluated to `false`, and (*b*) from passing executions of *mutated* versions of the program; for example, if flipping a branch predicate causes a failing test to pass, then the predicate was likely evaluated incorrectly in the original execution.

For each test case, let $\tau = t_1, t_2, \ldots, t_n$ denote the corresponding execution trace. If τ represents a failing execution, we mark the variables and values $v \in V_n^d$ created at the *failure point* t_n as being incorrect. As discussed earlier, when a function does not return gracefully (typically due to a program crash), the call statement t_i in its parent context is treated as the point of context close. In this situation, the variable node storing the return value at t_i should intuitively be considered incorrect. We thus additionally mark all variables $\{v \in V_i^d \mid \text{Close}(t_i) \wedge \text{Open}(t_i)\}$ as incorrect.

On the other hand, if τ corresponds to a passing execution, we mark all variables and values created throughout the trace, $v \in \bigcup_i V_i$, as being correctly evaluated. Although this lowers the suspiciousness of program statements executed in passing traces, recall that each statement still has a non-zero probability of being buggy given the fault tolerance built into our choice of CPDs.

Finally, we assume that all statements $\{s_i = \text{Stmt}(t_i) \mid t_i \in \tau\}$ in the test procedure are bug-free and therefore mark the corresponding statement vertices as correct.

Note 3.1. A more precise strategy may restrict correctness labels only to variables that appear in passing assertions. However, many real-world tests are *assertion-free*. In fact, across our entire benchmark dataset, 36% of the test cases do not contain assertions. In such cases, the absence of a failure during test execution is implicitly interpreted as a "*pass*". To account for such tests, we thus uniformly consider all variables and values created during passing executions to be correct. Another principled approach would be to ask the user to manually inspect and label the correctness of intermediate values. This approach is similar to those employed by Zhang et al. in Ursa [41] and

Xu et al. [38]. The main challenges we anticipate include determining which values to present for inspection, providing sufficient context for users to make informed judgments, and accommodating the possibility of user error. We believe this is an exciting direction for future research.

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 causes a failing test to pass, then that value is likely related to the bug. In particular, we focus on branch conditions because their domain consists of only two values, `false` and `true`.

After executing failing tests and constructing the EPG using the corresponding traces, we perform an initial Bayesian inference run to identify the $k = 20$ most suspicious runtime branch conditions. For each of these branch predicates c_r, we surgically alter the program to toggle the value of c_r at runtime and rerun the failing test to determine whether the test now passes. If the test outcome changes as a result of this intervention, then one or more of the following may hold in the original execution 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.

To account for both possibilities above, we introduce a new vertex `cfx` into the EPG. We also add the following edges to E: (a) $\langle \#@r \rangle \rightarrow$ `cfx`, and (b) $v \rightarrow$ `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 techniques do not consider the effect of modifying a statement on its dependent statements. In other words, they only adjust suspiciousness of the mutated statement in response to a successful mutation. Second, MBFL techniques are 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 [32], which translates Java bytecode into typed 3-address Jimple code. Our implementation consists of approximately 2,800 lines of Java code for instrumentation and static analysis, and 2,200 lines of Python code for EPG construction. We also use Bingo [30], which provides a convenient interface to LibDAI [24], for marginal inference.

As shown in Figure 4, PROSECUTOR uses FitSlicer [9], a test slicing tool, to extract an executable backward slice that preserves the failure-inducing behavior of the original failing test while eliminating statements in the test procedure that are irrelevant to the failure. This produces near-minimal failure-inducing tests, thereby reducing the size of the resulting execution traces and the EPGs.[1] We construct the EPG by executing these minimized failure-inducing tests and collecting the corresponding execution traces F. We also perform a coarse-grained coverage analysis to heuristically select a set of passing tests whose executed methods have substantial overlap with those in F. We then execute these tests to collect the corresponding set of useful passing traces P.

Ranking 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 $F \cup P$. As we will see in Section 5, running PROSECUTOR in mode F already

[1]The test slicing step does not affect the quality of the ranking, as statements irrelevant to the failure would otherwise be unreachable from the failure location in the EPG; instead, it preemptively eliminates a set of unnecessary vertices and edges.

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

1. For each program statement s that is a loop entry point, define:
$$\chi(s) = [i \mid \text{Stmt}(t_i) = s].$$
2. Over all loop entry points s, in decreasing order of number of occurrences, $|\chi(s)|$:
   A. Let $\chi(s) = [i^1, i^2, \ldots, i^m]$.
   B. For each $i^a \in \chi(s)$ such that $a + \gamma \leq m$, if IsLoop$(t_{i^a}, \ldots, t_{i^{a+\gamma-1}})$:
      a. Find the largest $c \geq a + \gamma$ such that IsLoop$(t_{i^a}, \ldots, t_{i^{c-1}})$. ▷ Can be done using Binary Search
      b. Let L be the identified loop period.
      c. Let $\sigma = t_{i^{c-2L}}, \ldots, t_{i^{c-1}}$ and $\tau' = t_1, t_2, \ldots, t_{i^{a-1}}, \sigma, t_{i^c}, \ldots, t_n$. ▷ Keep 2 repetitions of the loop
      d. Return Compress(τ', γ).
3. If $\gamma > 3$, return Compress$(\tau, \gamma - 1)$. ▷ No more loops with at least γ repetitions could be found
4. Return τ. No more loops could be found.

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 $r_s = \min(r_s^{F \cup P}, r_s^F)$ and by giving priority to $F \cup P$ to break ties. We will show in our experimental evaluation that this ranking aggregation achieves better performance than either approach alone.

Trace compression. Many test cases in Defects4J execute long-running loops, which increases the overhead of EPG construction and marginal inference. These loops, in fact, produce regular, 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 $t_i, t_{i+1}, \ldots, t_{i+\alpha}$ 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. The remaining iterations introduce the same dependencies as the last iteration. We therefore apply Algorithm 1 to the collected traces with an initial repetition threshold of $\gamma = 20$.

The implementation of Algorithm 1 relies on two key ideas: First, IsLoop can be evaluated for a trace of length n in linear time using the KMP prefix function [17] to determine periodicity. Second, during instrumentation at the Jimple IR level in Soot, we identify `goto` statements s_g together with their corresponding target statements s_t, and selectively insert annotations immediately before s_t whenever s_t`.lineNumber` $\leq s_g$`.lineNumber`. Such jumps target preceding statements within the same procedure and therefore correspond to backedges in the control-flow graph. These annotations allow us to efficiently identify loop entry points in the collected execution traces.

Alias analysis. 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(s), we consult Ref to determine if there is an aliasing variable w for each $v \in$ Def(s). In this case, we add w to Def(s) and transitively perform this augmentation until fixpoint. Note that we assume the aliasing relations remain persistent through the execution, which is guaranteed by the SSA-like format that Soot provides.

Dependency pruning. Finally, recall that the conditional probability distributions 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 when dealing with (a) invocation statements with several arguments, and (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 $v \in V_i^d$. 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 by 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 of possible fault locations?
RQ3. How do different aspects of EPG construction affect the quality of PROSECUTOR rankings?
RQ4. How do counterfactual executions affect the accuracy of fault localization?

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 [15]. These faulty versions contain, on average, 21K lines of source code. Furthermore, on average, each failing test consists of a trace of length 22,000 steps and executes 395 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 [26], Tarantula [14], DStar [34], and Ochiai [6],
(2) two MBFL approaches: MUSE [25] and Metallaxis [27], and
(3) two other state-of-the-art methods: SmartFL [37] and DepGraph [29], respectively.

We reimplemented SBFL and MBFL approaches using the Soot analysis framework [32] to get coverage profiles through instrumentation and by using Major [5] 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 and consulted the patches provided by Defects4J to derive the expected ground truth.

5.1.1 Quantitative Analysis. 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

Table 3. Effectiveness of Prosecutor and the baseline techniques in statement-level fault localization.

Project	Metric	DStar	Ochiai	Op2	Tarantula	Muse	Metallaxis	SmartFL	Prosecutor
Lang (58)	TOP-1	7	7	7	8	8	13	15	**20**
	TOP-3	20	20	20	21	16	26	30	**31**
	TOP-5	25	26	23	26	22	35	34	**41**
	TOP-10	39	38	39	38	29	38	42	**50**
	TOP-20	45	42	45	43	35	46	47	**56**
	EXAM	14.0	14.1	14.1	14.5	21.2	9.4	9.2	**5.7**
Compress (45)	TOP-1	10	9	10	8	2	9	6	**13**
	TOP-3	14	13	14	12	9	17	13	**19**
	TOP-5	19	17	19	16	12	18	17	**22**
	TOP-10	22	21	22	21	20	22	23	**29**
	TOP-20	23	22	23	23	22	26	27	**34**
	EXAM	14.4	14.8	14.4	14.4	20.0	9.2	10.3	**3.9**
Chart (26)	TOP-1	3	3	2	2	2	2	**5**	**5**
	TOP-3	7	7	6	7	3	5	14	**17**
	TOP-5	10	10	9	10	4	6	16	**22**
	TOP-10	11	12	10	12	7	8	18	**23**
	TOP-20	14	16	15	16	8	10	20	**25**
	EXAM	8.3	7.6	8.9	8.3	35.4	33.8	5.1	**2.0**
JacksonCore (25)	TOP-1	4	4	5	5	1	4	**7**	6
	TOP-3	8	7	8	9	3	6	10	**13**
	TOP-5	10	10	10	11	4	11	12	**14**
	TOP-10	12	11	12	13	6	12	13	**17**
	TOP-20	13	14	13	15	8	12	14	**18**
	EXAM	4.0	5.2	3.9	3.5	26.6	3.9	5.9	**2.3**
Math (82)	TOP-1	**15**	**15**	14	14	12	9	14	13
	TOP-3	25	25	23	25	17	20	**27**	**27**
	TOP-5	29	29	27	30	22	27	35	**38**
	TOP-10	35	37	33	37	31	35	39	**51**
	TOP-20	43	45	41	45	41	40	47	**57**
	EXAM	9.5	9.5	11.0	9.5	16.6	11.4	8.3	**5.5**
Jsoup (60)	TOP-1	7	7	5	7	3	4	5	**13**
	TOP-3	14	14	11	15	4	12	15	**20**
	TOP-5	20	19	18	18	6	15	19	**22**
	TOP-10	28	28	25	25	7	21	21	**29**
	TOP-20	34	**35**	31	32	10	23	25	34
	EXAM	**2.9**	**2.9**	3.7	3.4	41.9	44.8	23.7	3.4
Cli (39)	TOP-1	**9**	8	**9**	8	6	4	7	7
	TOP-3	**13**	12	12	11	7	10	11	10
	TOP-5	**18**	17	17	17	8	13	12	11
	TOP-10	**21**	**21**	20	20	9	17	13	18
	TOP-20	**24**	**24**	23	**24**	12	19	16	**24**
	EXAM	**8.0**	**8.0**	10.7	**8.0**	32.3	12.2	20.6	12.1

Project	Metric	DStar	Ochiai	Op2	Tarantula	Muse	Metallaxis	SmartFL	Prosecutor
Codec (16)	TOP-1	3	3	3	2	3	3	**5**	**5**
	TOP-3	5	5	5	5	4	4	**7**	6
	TOP-5	5	5	5	5	4	5	**8**	6
	TOP-10	7	7	7	7	6	8	**10**	9
	TOP-20	11	11	**13**	11	8	9	10	12
	EXAM	15.2	15.6	13.9	15.6	20.3	14.4	12.9	**10.7**
Mockito (38)	TOP-1	6	7	6	6	1	2	**9**	7
	TOP-3	12	13	11	10	3	5	13	**14**
	TOP-5	16	**18**	15	16	5	5	14	14
	TOP-10	19	**20**	17	19	5	7	15	18
	TOP-20	22	22	19	22	6	7	17	**26**
	EXAM	2.5	2.3	2.8	2.6	100	100	16.5	**1.6**
Time (26)	TOP-1	2	2	2	2	1	2	**4**	3
	TOP-3	5	4	5	5	2	5	**7**	5
	TOP-5	**10**	9	9	9	3	7	7	8
	TOP-10	**12**	**12**	11	11	4	7	10	11
	TOP-20	15	15	15	**16**	5	10	11	15
	EXAM	1.7	1.7	1.6	**1.2**	33.7	6.9	10.0	1.3
Csv (16)	TOP-1	4	4	4	3	2	3	4	**6**
	TOP-3	6	6	6	5	3	5	5	**7**
	TOP-5	**7**	**7**	**7**	**7**	4	**7**	6	**7**
	TOP-10	**8**	**8**	**8**	**8**	6	7	6	7
	TOP-20	**12**	**12**	**12**	**12**	7	8	7	9
	EXAM	**8.5**	**8.5**	**8.5**	**8.5**	30.0	24.0	23.0	15.8
Gson (17)	TOP-1	4	5	4	5	1	2	**8**	3
	TOP-3	7	7	7	8	3	3	**11**	8
	TOP-5	7	7	7	8	3	3	**11**	10
	TOP-10	**13**	11	10	11	3	3	12	10
	TOP-20	**14**	**14**	11	13	3	3	12	13
	EXAM	2.5	2.5	5.0	3.4	100	100	**1.9**	2.0
Collections (22)	TOP-1	5	5	5	5	n/a	n/a	**6**	3
	TOP-3	10	11	11	11	n/a	n/a	9	**13**
	TOP-5	14	15	14	15	n/a	n/a	11	**16**
	TOP-10	17	**19**	17	**19**	n/a	n/a	14	18
	TOP-20	18	19	17	19	n/a	n/a	18	**20**
	EXAM	7.0	5.8	6.2	5.8	n/a	n/a	11.1	**5.6**
Total (470)	TOP-1	79	79	76	75	42	57	95	**104**
	TOP-3	146	144	139	144	74	118	172	**190**
	TOP-5	190	189	180	188	97	152	202	**231**
	TOP-10	244	245	231	241	133	185	236	**290**
	TOP-20	288	291	278	291	165	213	271	**343**
	EXAM	7.7	7.3	8.4	7.1	32.3	19.5	10.3	**4.4**

within the top k entries of the reported rankings, and (b) the median EXAM score, which is the median percentage of code that must be examined to locate the faulty line across all benchmarks.

Overall, 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 fault locations can be identified by examining just 4.4% of the total lines of code covered by failing tests. We also include a visual presentation of these results for the entire dataset in Figure 10. Observe that the technique significantly outperforms *all* the baselines, and identifies 15% more bugs within its top-5 predictions than the best-performing one. Figure 10 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.

Figure 9 analyzes the overlap between our approach and each baseline. Across all comparisons for statement-level fault localization, observe that Prosecutor consistently localizes substantially more bugs than the baselines. Overall, the number of bugs uniquely localized by our approach (the dark green regions) is at least 2.3× greater than the number localized by each individual baseline technique (the red portions of the figure) and 3.5× greater than that of SmartFL. This further highlights the advantage of Prosecutor over the existing baseline techniques in localizing faults.

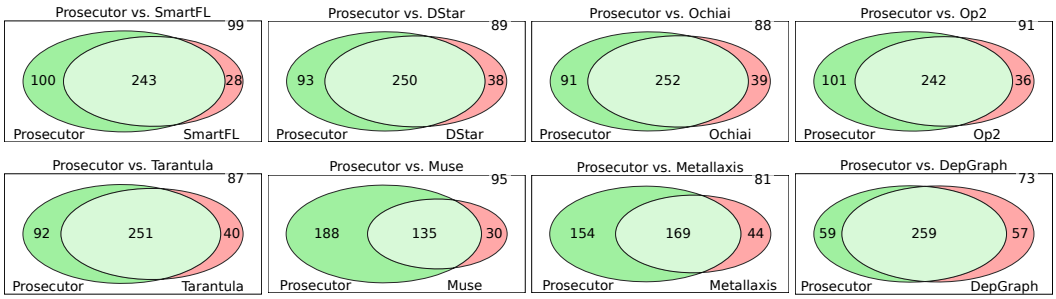


Fig. 9. Venn diagrams illustrating the overlap in faults localized by each pair of approaches, including faults localized by both methods, missed by both, or uniquely localized by either approach. Statement-level diagrams use a top-20 threshold to determine whether a bug is "localized" and the method-level diagram uses a top-3 threshold. Collections benchmarks are excluded from comparisons with Muse, Metallaxis, and DepGraph.

Our technique demonstrates strong improvements on bugs from 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 also 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 for future work.

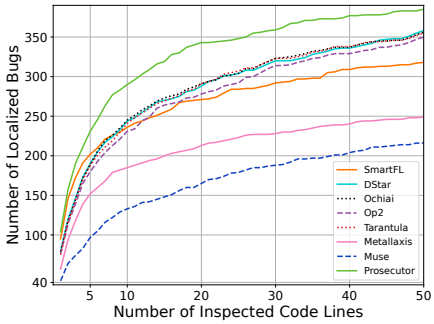


Fig. 10. Number of faults in top-k predictions of each technique across the entire dataset. The faster rising curves indicate higher effectiveness since an ideal technique would place all true fault locations as its first prediction.

Note 5.1. We could not run MBFL on Collections because Major does not support its required Java versions.

Comparison to learning-based techniques. We 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.

Observe that apart from three projects, Math, Cli, and Mockito, PROSECUTOR consistently matches or surpasses DepGraph in its top-2 through top-5 predictions for the most suspicious methods. Our approach even outperforms DepGraph at top-1 in several projects, including Csv and Compress. However, DepGraph generally demonstrates superior top-1 performance, which can be reasonably explained by its design focus on method-level fault localization, in contrast to statement-level localization which PROSECUTOR targets. Furthermore, DepGraph 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 could not run DepGraph on Collections benchmarks because of compatibility issues with the latest version of Defects4J. (*b*) Unlike PROSECUTOR, DepGraph does not consider the possibility of bugs being present in the test initialization methods. For a fair comparison, we also exclude these methods from the PROSECUTOR rankings when computing the statistics in Table 4.

Table 4. Comparison of PROSECUTOR and DepGraph in method-level fault localization.

Project	Method	TOP-1	TOP-2	TOP-3	TOP-4	TOP-5
Lang (58)	DepGraph	**44**	50	53	54	55
	PROSECUTOR	**44**	**54**	**56**	**56**	**56**
Compress (45)	DepGraph	23	30	32	35	37
	PROSECUTOR	**28**	**35**	**36**	**39**	**39**
Chart (26)	DepGraph	**16**	19	20	21	21
	PROSECUTOR	12	**21**	**24**	**24**	**25**
JacksonCore (25)	DepGraph	**10**	13	15	15	15
	PROSECUTOR	8	**15**	**17**	**18**	**19**
Math (82)	DepGraph	**54**	**66**	**71**	**76**	**76**
	PROSECUTOR	43	55	60	67	72
Jsoup (60)	DepGraph	18	**25**	28	29	32
	PROSECUTOR	**19**	**25**	**32**	**33**	**34**
Cli (39)	DepGraph	**16**	**21**	**22**	**27**	**28**
	PROSECUTOR	10	15	17	21	23
Codec (16)	DepGraph	**8**	**10**	10	10	11
	PROSECUTOR	**8**	**10**	**11**	**14**	**14**
Mockito (38)	DepGraph	**18**	**24**	**27**	**30**	**31**
	PROSECUTOR	17	22	25	27	27
Time (26)	DepGraph	**14**	**16**	**17**	**17**	17
	PROSECUTOR	9	**16**	16	16	**18**
Csv (16)	DepGraph	6	9	9	**10**	**11**
	PROSECUTOR	**9**	**10**	**10**	**10**	**11**
Gson (17)	DepGraph	**11**	11	12	12	13
	PROSECUTOR	5	**13**	**14**	**14**	**14**
Collections (22)	DepGraph	n/a	n/a	n/a	n/a	n/a
	PROSECUTOR	6	14	17	18	21
Total (470)	DepGraph	**238**	294	316	336	347
	PROSECUTOR	218	**305**	**335**	**357**	**373**

5.1.2 Qualitative Analysis. We also conducted a qualitative analysis to identify scenarios in which PROSECUTOR outperforms SmartFL and vice versa. We now elaborate on such cases through a series of representative examples.

Note that SmartFL creates its probabilistic graph by parsing Java bytecode traces and simulating their execution using an operand stack, where pushing a variable onto the stack is treated as a definition and popping it is treated as a use. In contrast, PROSECUTOR does not rely on stack-based execution to construct the EPG. Instead, our EPG is built upon 3-address statements which are executed along the trace given the inference rules in Figures 5, 7, and 8 and the Def/Use sets in Table 2. The following examples clarify the fundamental differences in our graph construction.

Example 2. Consider the test case in Figure 11 which fails on Line 12 since the variable p2 is incorrectly computed as a result of a bug within the method combine. When processing the trace and the method call at Step 5, PROSECUTOR derives an edge from the vertex ⟨p2@4⟩ to the vertex ⟨subs@5⟩. Other similar edges in the EPG will be as follows: ⟨p2@1⟩ → ⟨p2@4⟩, ⟨p3@2⟩ → ⟨p3@3⟩, and ⟨p3@3⟩ → ⟨p2@4⟩. These "*shortcut*" edges are derived by Rule R_{DU} in Figure 5 given the Def/Use sets defined for invocation statements in Table 2.

```
1  public ExtendedProperties subset(String prefix) {
2    ExtendedProperties c = new ExtendedProperties();
3    ... /* several statements inside the method */
4    return c;
5  }
6  public void testCase() {
7    ExtendedProperties p2 = new ExtendedProperties();
8    ExtendedProperties p3 = new ExtendedProperties();
9    p3.setProperty("sub.test", "foo");
10   p2.combine(p3);
11   ExtendedProperties subs = p2.subset("sub");
12   assertNotNull(subs);
13 }
```

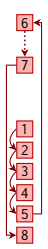


Fig. 11. Test snippet adapted from the Apache Commons Collections library [1], illustrating the bug in version #12 of this project in Defects4J. The program contains a bug in the combine method, resulting in an incorrect update to the variable p2 on Line 10 and an assertion violation on Line 12.

In contrast, upon processing the method call on Line 11, SmartFL pushes the value of p2 onto the operand stack and creates a corresponding node. Whenever this value is popped from the stack within the subset method, SmartFL introduces an edge from this vertex to the point of use. The value popped from the stack may be used to define a second variable which itself can influence a third variable and so on. Observe that this chain of dependencies may grow arbitrarily long and involve multiple call frames, eventually propagating to and affecting the return value of the method subset and therefore the value of the subs object.

In practice, an incorrect value which is returned/created as a result of a bug within a method (in our case, the combine method on Line 10) may be passed as an input argument or used as the base object in subsequent calls, where it can propagate and taint other values. Still, as long as the value remains structurally valid, subsequent methods may process it without issue (e.g., the method call

on Lines 11), and the error typically surfaces only when the value becomes invalid or is explicitly checked by an assertion. The graph representation which is used by SmartFL performs poorly in such scenarios, where the actual fault resides in higher-level methods far from the failure location.

As a consequence, while PROSECUTOR surfaces this bug at Rank 5, its position in SmartFL is lower, at Rank 10. We perform a more comprehensive quantitative analysis of the effect of these shortcut edges in our ablation study in Section 5.3.3. In particular, removing shortcut edges in this example (using the alternative Rule R'_{DU}) causes the actual fault to drop to Rank 8 in the PROSECUTOR ranking.

The EPG also introduces an additional set of shortcut edges that more effectively model error propagation when the failure occurs within a nested callee and results in an abrupt program termination while multiple frames remained unpopped on the call stack. As discussed earlier, regardless of whether a method terminates gracefully, Rule R_{CL1} derives edges from vertices defined by context-closing statements to those in V_j^d, where t_j is the call statement in the parent context.

Example 3. Consider the program in Figure 12 with a bug on Line 3 whose execution leads to an unexpected method invocation at Step 3. When constructing the EPG for this execution, our inference rules specifically derive the following two edges: (*a*) Rule R_{CL1} derives an edge from the vertex ⟨#@7⟩ corresponding to the thrown exception to the variable vertex ⟨`logText`@3⟩, and (*b*) Rule R_{CTR1} derives an edge from the value vertex ⟨#@2⟩ to the vertex ⟨`logText`@3⟩.

```
1  public static void processApplicationLog() {
2    boolean ready = writeApplicationLog();
3    if (!ready) { // faulty line
4      String logText = readApplicationLog();
5      ... /* several statements inside the method */
6    }
7  }
8  public static String readApplicationLog() {
9    try {
10     Path path = new File(LOG_FILE).toPath();
11     String content = Files.readString(path);
12     return content;
13   } catch (IOException e)
14   { throw new RuntimeException(e.getMessage()); }
15 }
```

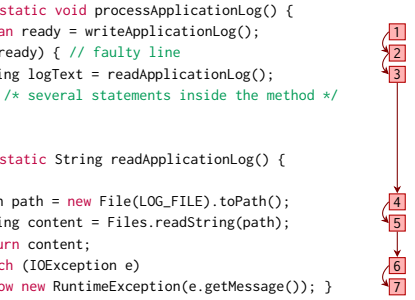


Fig. 12. Example method with a bug on Line 3 which should instead be `if(ready)`. The incorrect condition causes the program to crash on Line 14.

Although the variable `logText` is never instantiated at runtime, we still include the corresponding vertex in the EPG because our EPG construction operates on 3-address statements rather than on the concrete stack execution. In addition to the value vertex ⟨#@7⟩ at failure point, we also mark the variable vertex ⟨`logText`@3⟩ as being incorrect since, at a minimum, the method `readApplicationLog` must return gracefully for the value `logText` to have any chance of being correct. The observation ¬⟨`logText`@3⟩ along with the edges (*a*) and (*b*) allows the model to assign a high suspiciousness score to the `if`-statement on Line 3.

In contrast to PROSECUTOR, SmartFL does not place Line 3 among its top predictions within the ranking. This occurs since the predicate corresponding to this line is not reachable from the failure point in the underlying graph and therefore the error does not propagate appropriately. SmartFL's graph construction assumes that the callee procedure consistently uses the arguments passed at the call site, and these data dependence edges enable the propagation of errors to top-level procedures. Of course, if such arguments existed, the corresponding vertices would be pushed onto the stack under the control of the predicate on Line 3. If these values were later popped within the method `readApplicationLog` and contributed to error propagation, then the statement on Line 3 would be reachable from the failure location in SmartFL's graph. However, in the case of the program in Figure 12, where the callee method `readApplicationLog` receives no arguments, this graph construction fails to connect two consecutively executed methods, even though the root cause of the fault resides in the caller procedure.

The above example simply demonstrates how PROSECUTOR more effectively localizes faults in higher-level procedures by incorporating shortcut edges that connect consecutive incomplete call frames. In a similar way, PROSECUTOR ranks the bugs in versions #16 and #41 of the Compress project [2] at Rank 1, whereas SmartFL places them at Ranks 62 and 358 of its ranking, respectively.

Example 4. Finally, our constructed EPG also models the effect of unexplored branches on the executed statements. Consider the example code in Figure 13 where given two input strings that represent two integers with different number of digits, the method `getMinStr` produces the integer value for the number with fewer digits. Given the failing execution in this figure, our inference rules derive and include the following edges in the EPG: ⟨#@8⟩ → ⟨str@9⟩, ⟨val@11⟩ → ⟨res@9⟩, and ⟨#@8⟩ → ⟨res@9⟩. Similar to the previous example, we also incorporate the observations ¬⟨res@9⟩ and ¬⟨val@11⟩ into the model which allows it to place the `if`-statement on Line 10 within its top predictions for the bug location.

```
1  public static Integer getMinStr(String a, String b) {
2    if (a == null || b == null)
3      throw new IllegalArgumentException("Null Args");
4    int a_len = a.length();
5    int b_len = b.length();
6    int min = Math.min(a_len, b_len);
7    if (min > 10 || a_len == b_len)
8      throw new IllegalArgumentException("Invalid Args");
9    String s = a;
10   if (b_len > a_len) // faulty line
11     s = b;
12   Integer res = createInteger(s);
13   return res;
14 }
15 public static Integer createInteger(final String str) {
16   if (str == null)
17     return null;
18   Integer val = Integer.decode(str); // crash
19   return val;
20 }
21 public void testCase() {
22   Integer x = StringUtils.getMinStr("12345678912", "12");
23   assertEquals(12, x);
24 }
```

Fig. 13. Example string processing method with a bug on Line 10, which should be `if (b_len < a_len)`.

On the other hand, none of these edges are included in SmartFL's probabilistic graph since (*a*) SmartFL does not consider the effect of unexplored branches which we handle by Rules R_{OP3}, R_{OP4}, and R_{CTR2}, and (*b*) it does not model the effect of callee statements on the caller statements when the callee procedure terminates ungracefully due to its reliance on actual stack execution.

This example demonstrates how Prosecutor achieves more accurate bug localization (as a result of (*a*)) in Jsoup version #39, placing the actual faulty statement at Rank 9, while SmartFL places it at Rank 628. At the same time, it demonstrates how considering (*b*) enables Prosecutor to localize the fault in version #1 of the Lang project at Rank 2 while SmartFL places this bug at Rank 8.

Example 5. In contrast to the example in Figure 11, note that the set of shortcut edges we draw in the EPG can occasionally have a negative effect on the fault ranking. Consider the code in Figure 14 with a bug in the method `absURL`. In this example, our inference rules include the following two edges in the EPG: ⟨a1@2⟩ → ⟨str@3⟩ and ⟨#@6⟩ → ⟨str@3⟩. Notice that SmartFL's graph does not include the edge ⟨a1@2⟩ → ⟨str@3⟩. Since the actual fault resides on the path reachable by tracing backward through the edge ⟨#@6⟩ → ⟨str@3⟩, SmartFL ranks the fault at Rank 2, while Prosecutor spreads suspicion somewhat between the two paths and—after Bayesian inference—ranks the actual fault at Rank 7. Removing shortcut edges in this case can also raise the fault's position in the Prosecutor ranking to Rank 4 instead.

```
1  public String absUrl(String attributeKey) {
2    ... /* several statements inside the method */
3    URL abs = new URL(base, relUrl); // faulty line
4    String res = abs.toExternalForm();
5    return res;
6  }
7  public void testCase() {
8    Document doc = Jsoup.parse(
9    "<a href='?fo'>One</a> <a href='bar.html?fo'>Two</a>",
10   "http://jsoup.org/path/file?bar"
11   );
12   Element a1 = doc.select("a").first();
13   String str = a1.absUrl("href")
14   assertEquals("http://jsoup.org/path/file?fo", str);
15 }
```

Fig. 14. Test snippet adapted from the Jsoup parsing library [4], illustrating the bug in version #10 of this project in Defects4J. The code contains a bug on Line 3, which also results in an incorrect value being returned.

Overall, note that SmartFL does not involve several types of edges which we consider to make fault localization more general and effective across diverse scenarios. In benchmarks where the bug can be captured only through simple data and control dependencies—where SmartFL's modeling is already sufficient—the additional edges introduced in the EPG may occasionally have a slight

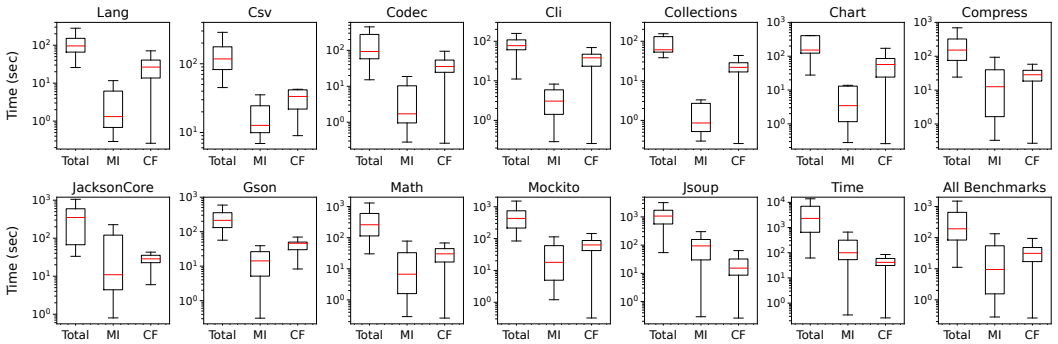


Fig. 15. PROSECUTOR running time breakdown, including the total time and the time needed for performing marginal inference (MI) and counterfactual analysis (CF).

negative impact on the ranking results. However, as we demonstrate in our ablation study, these edges significantly improve the generality of PROSECUTOR. With all edges included, PROSECUTOR localizes 73% of the faults within the top-20 entries of its ranking, while removing shortcut edges at return sites and call sites reduces this number to 67% and 69%, respectively.

In summary, our qualitative analysis shows that PROSECUTOR generally performs better when:

(1) errors propagate across method boundaries through arguments or receiver objects, resulting in a long propagation chain between the point of error introduction and the failure location;
(2) program execution terminates abruptly due to a crash or exception within nested callees; and
(3) an unexplored branch prevents desirable updates to a value which is involved in the failure.

On the other hand, SmartFL occasionally performs better when (1) failing tests execute only a few methods (i.e., one or two methods) without encountering a crash, and (2) errors propagate only through standard data and control dependencies. In such cases, our additional EPG edges may dilute the suspiciousness of actual fault locations, thereby placing them lower in the ranking.

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 253 seconds to localize each fault (using geometric means). EPG construction (56%), marginal inference (16%) and counterfactual analysis (5%) are the main contributors to this running time. Figure 15 shows the breakdown of these statistics across different projects.

We observed that SmartFL and SBFL are the most lightweight approaches, requiring on average 67 and 76 seconds per benchmark, respectively. Unsurprisingly, MBFL is the most expensive technique, requiring 765 seconds per benchmark on average. Overall, our approach is approximately 3× slower than SmartFL. Most of this gap is due to the graph-construction step, for which SmartFL uses a faster parallelized implementation. In principle, we can also parallelize EPG construction, which we expect would yield similar speedups.

Our implementation of PROSECUTOR achieves an average graph-construction throughput of 70 stmt/sec, and the loop-identification algorithm processes traces at an average rate of 695 stmt/sec. Furthermore, our experiments show that loop reduction eliminates more than 38,000 statements from the processed traces per benchmark on average. With loop compression turned off, PROSECUTOR requires an average of 489 seconds per benchmark to produce its ranking, which is approximately 2× the time needed with loop compression enabled. This clearly highlights the importance of loop identification and compression to the overall scalability of the procedure.

5.3 RQ3: Ablation Study

5.3.1 Effect of Passing and Failing Traces. We ran Prosecutor in two ablated settings to measure the impact of failing and passing traces on the resulting ranking quality. To this end, we eliminate the ranking aggregation phase and separately evaluate the rankings obtained by constructing the EPG from only failing traces, F, and from both failing and passing traces, $F \cup P$.

Table 5. Results of Prosecutor in various ablated settings. The rows labeled with F and $F \cup P$ show the performance without ranking aggregation, and the subsequent rows present the results without loop compression, shortcut edges, control dependence edges and counterfactual analysis, respectively.

Setting	TOP-1	TOP-3	TOP-5	TOP-10	TOP-20	EXAM
Prosecutor	104	**190**	231	**290**	**343**	**4.4**
F	99	183	218	277	324	4.8
$F \cup P$	102	189	229	282	326	5.0
w/o Loop Compression	102	187	229	288	338	4.6
w/o Shortcut (Call site)	**107**	189	**235**	285	328	5.0
w/o Shortcut (Return site)	85	161	202	263	317	5.5
w/o Ctrl Dependencies	86	169	204	262	302	5.9
w/o Counterfactual	104	187	222	280	332	4.7

We report these statistics using the top-k measure in Table 5. Comparing Tables 3 and 5 shows that even the first setting, which does not leverage passing traces, outperforms the best-performing baseline in top-k predictions for all $k \in \{1, 3, 5, 10, 20\}$. Note also that, on average, only two failing tests trigger the failure in each Defects4J benchmark. Therefore, Prosecutor can still effectively localize faults using only a small number of failing tests without requiring passing traces. This represents a significant advantage over SBFL techniques, with which effective fault localization is essentially impossible without passing executions. Nevertheless, incorporating information from passing traces into the EPG further improves the quality of the resulting rankings.

5.3.2 Effect of Loop Compression. Given that our loop compression algorithm eliminates redundant statements (and therefore redundant dependencies) from execution traces, we also conducted an ablation study to assess its potential impact on the overall accuracy of Prosecutor. As shown in Table 5, loop compression does not compromise the accuracy of Prosecutor; instead, it slightly improves the resulting rankings while significantly enhancing the overall efficiency of the system.

5.3.3 Effect of Shortcut Edges. We also conducted two additional ablation studies in which we disable the shortcut edges described in Section 5.1.2 to assess their impact on ranking accuracy.

In the first experiment, we disable Rule R_{CL1} when t_i is not a `return` statement. In this setting, we also exclude observations on the incorrectness of $\{v \in V_i^d \mid \text{Close}(t_i) \wedge \text{Open}(t_i)\}$, where v represents the destination vertices of these shortcut edges.

In the second experiment, we restrict Rule R_{DU} to non-invocation statements. Alternatively, one can modify Rule R_{DU} as follows, thereby excluding shortcut edges that abstractly capture the potential impact of method arguments and the base object on return values at call sites:

$$R'_{\text{DU}}\ \frac{v \in V_i^d \quad u \in \text{Use}(s_i) \quad \neg\,\text{IsRcvObj}(u, t_i) \quad \neg\,\text{IsArg}(u, t_i) \quad \langle u@j\rangle = \text{LastDef}(u, t_i)}{\langle u@j\rangle \rightarrow v \in E}$$

Once again, Table 5 presents the results of these experiments. Overall, removing the shortcut edges in our first experiment decreases the top-1 and top-3 metrics by 18% and 15%, respectively. As described earlier, these edges are specifically introduced by Rule R_{CL1} when execution fails within nested callee procedures while several call frames remain active on the execution stack (the examples in Figures 1, 12, and 13 illustrate such cases). These failures, which mainly occur in the form of unexpected exceptions or program crashes, lead to abrupt execution termination. Our measurements show that, among the 958 failure-triggering tests in our benchmark set, 437 (46%) fail before reaching an assertion in the corresponding test procedure. Since such cases account for a substantial fraction of the failing tests, disabling shortcut edges at return sites naturally results in

a noticeable decline in ranking accuracy. Observe also that removing shortcut edges at invocation sites has only a limited impact on the top-10 metric but still significantly affects the top-20 metric.

5.3.4 Effect of Control Dependence Edges. As our final ablation study, we exclude control dependence edges from the EPG to evaluate the contributions of data and control dependencies to fault localization performance. Specifically, we disable Rules R_{CTR1}, R_{CTR2}, R_{OP3}, and R_{OP4} when constructing the EPG from execution traces. Note that, in this setting, we still preserve the value vertices representing the correctness of branch predicates evaluated along the traces. We aggregate the results under this setting and report them in Table 5.

Without control dependence edges, the top-1 and top-5 metrics drop by 17% and 12%, respectively, highlighting the importance of these edges in the EPG. Note also that excluding these edges may naturally improve the rankings for bugs that only affect data along a failing execution. This occurs because removing certain paths between vertices in the EPG may reduce the set of statements reachable from the failure location and, in turn, shrink the search space of candidate fault locations.

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 this feature disabled. Table 5 also reports the results of this study. Overall, observe that the number of faults identified within the top-k entries drops when counterfactual observations are excluded.

In particular, we observed that in 36% of the benchmarks, at least one counterfactual experiment changes the eventual test outcome. This clearly shows that the initial Bayesian inference can effectively prioritize suspicious branch conditions potentially related to the fault, which again highlights the effectiveness and precision of our modeling. In such cases, counterfactual analysis improves the bug rank by 5 positions on average. Overall, 77% of bugs are either positively affected or remain unchanged. For positively affected bugs, their positions in the ranking improve by 30 positions on average, whereas negatively affected bugs move down by only 8 positions on average. Furthermore, successful counterfactual experiments increase the number of faults identified within the top-5 and top-10 predictions by 9%.

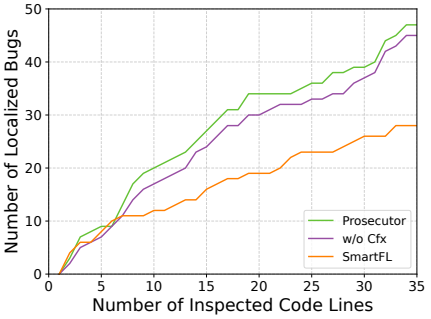


Fig. 16. Number of deep faults in top-k predictions of PROSECUTOR–with and without counterfactual analysis–against SmartFL. The benchmarks contain 86 deep faults in total.

We specifically observed that counterfactual analysis helps prioritize deep bugs in the ranking. While the median distance of buggy statement vertices from the failure location in the EPG is 7 edges, this distance increases to 10 edges in cases where counterfactual analysis improves the ranking. At a high level, program elements farther from the failure point in the EPG typically appear less suspicious. In contrast, marking the virtual variable `cfx` as incorrect directs suspicion toward statement nodes with edges leading to the `cfx` vertex, as well as 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 56 in the ablated setting, while the full PROSECUTOR places this statement at Rank 16. This evidently shows the effect of counterfactual analysis in prioritizing deep bugs.

A broader view of this improvement across all benchmarks containing deep bugs is shown in Figure 16. We specifically define the depth of a bug as the number of edges in the EPG between the buggy statement and the failure location. We also consider bugs whose depths are at or above

the third quartile (i.e., ≥ 11 edges) as "*deep bugs*". Compared to SmartFL, Prosecutor without counterfactual analysis can localize 42% and 58% more deep faults within its top-10 and top-20 predictions, respectively. Incorporating counterfactual analysis into the method further enhances the accuracy of localization and allows our approach to identify 67% and 79% more deep bugs than SmartFL within its top-10 and top-20 predictions, respectively.

Finally, beyond improvements in the top-k metric, counterfactual analysis serves as an additional and meaningful source of validation. Even when the true fault location is already among the top-ranked predictions before counterfactual analysis, a successful counterfactual experiment further increases the suspiciousness of the faulty statement, reinforcing confidence in its ranking. Notably, we observed that incorporating counterfactual results back into the model alters the suspiciousness of buggy statements, on average, 5× more than that of non-buggy statements.

6 Limitations of Our Approach

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

First, our approach primarily targets "*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 are never raised. We handle such cases by performing a lightweight analysis to identify and annotate possible failure points where the missing exception could have been raised. A more general solution to this problem remains an interesting direction for future work.

Next, recall that the EPG models how errors arise and propagate during program execution. Although it captures a large number of errors occurring in real-world programs, our inference rules are not sound and may fail to capture complex aliasing patterns (our current solution relies on static specifications to identify potentially aliasing statements) or adequately model heap dependencies. More precise modeling of data dependencies may therefore improve the overall effectiveness of Prosecutor. However, soundly capturing additional sources of error propagation may also introduce spurious dependencies and ultimately reduce ranking quality, reflecting a tradeoff similar to the precision-recall tradeoffs commonly encountered in program analysis and machine learning.

Finally, our current implementation faces difficulties 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 trace compression. Note that the EPG construction algorithm makes a single pass over an execution trace with n statements, thereby requiring $O(n)$ time. The main performance bottleneck instead lies in the I/O overhead of exchanging data between the Java and Python processes. Besides parallelizing graph construction, performing static dependency analysis offline could significantly reduce this overhead and further improve scalability. Another interesting direction for future work is to identify and prune unnecessary dependencies in the EPG, making 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 [35].

Fault localization. The introduction of delta debugging for simplifying and isolating failure-inducing inputs [39] was perhaps among the earliest works to spark a line of research on 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 [6, 7, 14, 26, 34]. This line of research was later extended by mutation-based techniques, which inspect test behavior across a set of mutants to assess the impact

of changing each statement on test outcomes [11, 25, 27]. Related approaches also explore how intervening on program execution affects observed failures [28, 42]. Predicate switching [42] alters control flow by heuristically identifying critical predicates and inverting their outcomes during failing runs, which may be regarded as a simple form of counterfactual analysis. More recently, Flex [28] uses counterfactual executions to improve spectrum-based fault localization. In contrast, our work describes how to effectively select a set of useful counterfactual experiments and how to incorporate their results into a single unified probabilistic model.

Beyond techniques that reason primarily from test executions, history-based techniques prioritize fault candidates based on historical defect data [16, 31]. Such data can also be combined with coverage profiles to improve fault localization accuracy [33].

Besides statistical approaches, learning-based fault localization has recently gained significant attention. These techniques apply learning to different aspects of the fault localization process. UniVal [19] uses statistical causal inference to estimate the average causal effect of counterfactual assignments to program variables without actually executing the program. Neural-MBFL [13] uses a pre-trained model to leverage contextual information surrounding mutation locations and improve the quality of generated mutants. GMBFL [36], in turn, models relationships among code entities, mutants, and test runs using a graph representation and employs gated graph attention neural networks to learn useful features from these relationships.

Another line of work applies learning-based techniques to method-level fault localization. DeepFL [21] combines suspiciousness scores from SBFL and MBFL with code-complexity and textual-similarity features in a unified deep learning model. More recent approaches explore richer representations of program structure and test behavior. Grace [22] leverages graph-based representation learning to capture coverage relationships between tests and program entities. DepGraph [29] proposes a more compact representation by incorporating the inter-procedural call graph into a GNN model, along with code-change information as an additional feature. HetFL [10] addresses data imbalance and representation limitations of previous approaches by resampling faulty data and modeling programs as heterogeneous graphs.

These techniques, however, perform only method-level localization, providing limited discrimination among candidate fault locations when failing unit tests exercise only a small number of methods. In our experiments, failing tests in the Defects4J dataset execute a median of 19 and an average of 30 methods before failure. In such cases, identifying the faulty method alone may provide limited diagnostic value, as developers must still inspect potentially many statements within the reported methods to locate the fault. In addition, our technique requires neither training data nor a GPU, making it more broadly applicable and less expensive than most learning-based approaches.

Probabilistic program reasoning. Starting with Eugene [23] and Bingo [30], a growing body of work has explored the use of Bayesian inference for probabilistic reasoning about programs. Xu et al. [38] model debugging as a probabilistic inference problem in which human-like reasoning rules are encoded as conditional probability distributions. Inspired by this work, SmartFL [40] constructs an efficient probabilistic model of program semantics using dynamic data and static control dependencies. While it suggests a promising direction for effective fault localization, SmartFL's encoding of program semantics is subject to several limitations, as discussed throughout this paper.

8 Conclusion

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

Data Availability Statement

The artifact accompanying this paper is publicly available on Zenodo [8]. This artifact contains the implementation of Prosecutor and the baseline techniques, along with the scripts to reproduce the experimental results. The OOPSLA-2026-AEC-submission version was submitted for artifact evaluation and the OOPSLA-2026-AEC-final version reproduces the final results of our paper.

Acknowledgements

We thank the anonymous reviewers for their constructive feedback. This research was supported in part by the U.S. National Science Foundation (NSF) under grants CCF-2146518 and CCF-2313054.

References

[1] [n. d.]. *Apache Commons Collections Java Library.* https://commons.apache.org/proper/commons-collections
[2] [n. d.]. *Apache Commons Compress Java Library.* https://commons.apache.org/proper/commons-compress
[3] [n. d.]. *Apache Commons Lang Java Library.* https://commons.apache.org/proper/commons-lang
[4] [n. d.]. *Jsoup: Java HTML Parser.* https://jsoup.org
[5] [n. d.]. *The Major Mutation Framework.* https://mutation-testing.org
[6] 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 (2009), 1780–1792. doi:10.1016/j.jss.2009.06.035
[7] Rui Abreu, Peter Zoeteweij, and Arjan van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 88–99. doi:10.1109/ASE.2009.25
[8] Sara Baradaran, Yifei Huang, Wei Le, and Mukund Raghothaman. 2026. *Prosecutor: Bayesian Counterfactual Fault Localization (Artifact)*. doi:10.5281/zenodo.21941781
[9] Sara Baradaran and Mukund Raghothaman. 2026. Effective Minimization of Failure-Inducing Tests Using Convention-Aware Slicing. In *IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*. 349–356. doi:10.1109/SANER-C67878.2026.00053
[10] 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
[11] 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
[12] 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 (1991), 451–490. doi:10.1145/115372.115320
[13] 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 *IEEE Annual Computers, Software, and Applications Conference (COMPSAC)*. 1274–1283. doi:10.1109/COMPSAC61105.2024.00168
[14] James Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, 273–282. doi:10.1145/1101908.1101949
[15] 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
[16] 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
[17] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. 1977. Fast Pattern Matching in Strings. *SIAM journal on computing* 6, 2 (1977), 323–350. doi:10.1137/0206024
[18] Daphne Koller and Nir Friedman. 2009. *Probabilistic Graphical Models: Principles and Techniques*. The MIT Press.
[19] 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
[20] Tom Leonard, John S. J. Hsu, and Kam-Wah Tsui. 1989. Bayesian Marginal Inference. *J. Amer. Statist. Assoc.* 84, 408 (1989), 1051–1058. doi:10.1080/01621459.1989.10478871
[21] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. DeepFL: Integrating Multiple Fault Diagnosis Dimensions for Deep Fault Localization. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 169–180. doi:10.1145/3293882.3330574

[22] 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 (ESEC/FSE)*. ACM, 664–676. doi:10.1145/3468264.3468580

[23] Ravi Mangal, Xin Zhang, Aditya V. 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. doi:10.1145/2786805.2786851

[24] 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

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

[26] 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 (2011). doi:10.1145/2000791.2000795

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

[28] Jongchan Park, Tae Eun Kim, Dongsun Kim, and Kihong Heo. 2025. Improving Fault Localization with External Oracle by Using Counterfactual Execution. *ACM Transactions on Software Engineering and Methodology* 34, 2, Article 35 (2025). doi:10.1145/3695997

[29] 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. *Proceedings of the ACM on Software Engineering* 1, FSE, Article 86 (2024). doi:10.1145/3660793

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

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

[32] 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. doi:10.1145/1925805.1925818

[33] 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

[34] 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

[35] 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. doi:10.1002/9781119880929.ch1

[36] 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

[37] Yiqian Wu, Yujie Liu, Yi Yin, Muhan Zeng, Zhentao Ye, Xin Zhang, Yingfei Xiong, and Lu Zhang. 2025. SmartFL: Semantics Based Probabilistic Fault Localization. *IEEE Transactions on Software Engineering* 51, 7 (2025), 2161–2180. doi:10.1109/TSE.2025.3574487

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

[39] 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

[40] 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

[41] Xin Zhang, Radu Grigore, Xujie Si, and Mayur Naik. 2017. Effective Interactive Resolution of Static Analysis Alarms. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 57 (2017). doi:10.1145/3133881

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

Received 2026-03-17; accepted 2026-06-10