A Metric for Measuring the Impact of Rare Paths on Program Coverage

Leo St. Amour
Dept. of Computer Science
Virginia Tech
Blacksburg, USA
lstamour@vt.edu

Eli Tilevich
Dept. of Computer Science
Virginia Tech
Blacksburg, USA
tilevich@cs.vt.edu

Muhammad Ali Gulzar
Dept. of Computer Science
Virginia Tech
Blacksburg, USA
gulzar@cs.vt.edu

***Abstract*—Fuzzing has become a popular technique for discovering bugs and vulnerabilities. To increase the probability of finding bugs, developers should apply fuzzers that maximize program coverage. Program coverage typically measures the percentage of program lines or branches a fuzzer executes. However, these metrics fail to communicate the value of hitting a particular line, branch, or path. Many bugs manifest only within non-trivial control flows. To improve software quality, fuzzing non-trivial program paths should be more important than fuzzing trivial ones. This paper introduces *rare-path coverage (RP-Coverage)*, a novel program coverage metric that conveys the value of discovering an unlikely control flow path. We have developed a new technique for estimating the probability of taking an execution path, which relies on probabilistic logic programming to declaratively express the logic for constructing and analyzing a probabilistic control flow graph. Our evaluation indicates RP-Coverage's promise as a metric for measuring fuzzing efficacy. Specifically, we observe that defects along rare paths—intuitively—substantially impact the effectiveness of fuzzers. However, we argue that existing fuzzing metrics fall short when conveying this significance. We also observe that the value of uncovering an unlikely path is better reflected by increases in RP-Coverage than existing metrics. Specifically, the average coverage increases are up to 49.5%, 11.1%, and 15.4% for RP-Coverage, line coverage, and branch coverage, respectively. This finding indicates that RP-Coverage is more elastic, or sensitive, to path probabilities and thus capable of more effectively quantifying a fuzzer's ability to discover unlikely program paths. As such, RP-Coverage demonstrates promise as a program coverage metric that enhances fuzzer fitness measures when supplementing standard criteria.**

***Index Terms*—Analysis metrics, Program coverage, Static analysis, Fuzzing**

I. Introduction

> Two roads diverged in a wood, and I – I took the one less traveled by, and that has made all the difference.
>
> *Robert Frost*

Software defects, omnipresent in any non-trivial code base, threaten the system's safety and security. Software maintenance is concerned with improving software quality by detecting and eliminating these defects before or after they are discovered in a live environment [1]. However, effective defect detection remains an open problem. While some defects are triggered during typical program execution, others manifest along an uncommon subset of the program's semantics and occur under atypical execution scenarios.

Consider the program depicted in Figure 1, which executes an important function only if `get_rand_letter` returns a lowercase or uppercase "z". This program contains two defects. Defect ❶ causes the program to crash if it is not provided with a command line argument. This bug exemplifies a common defect, likely to be captured in a test suite or discovered through routine program use. Defect ❷ results from an off-by-one error on line four. If `get_rand_letter` returns a lowercase "z", the important function does execute. This bug exemplifies a rare defect. The unintended behavior manifests only for one of the potential 52 values that `get_rand_letter` can return, suggesting that this bug has a 1/52, or less than 2%, probability of triggering.

```
   1  int main(int argc, char *argv[]) {
❶ 2      char *arg = argv[1];
   3      char c = get_rand_letter();
   4      if (97 <= c && c <= 121) {
   5          c = make_uppercase(c);
   6      }
   7      if (c == 'Z') {
❷ 8          important_function();
   9      }
  10      ...
  11  }
```

Fig. 1: Common and rare program defects

The coverage achieved by a fuzzer and its ability to find bugs is strongly correlated [2]. To improve mutation-based fuzzing, recent research efforts focus on increasing program coverage by biasing fuzzers towards "rare" execution paths [3], [4]. Nevertheless, existing coverage criteria may not be able to fully convey the value these efforts contribute to fuzzer performance. Consider Fuzzer A and Fuzzer B, which cover 50% and 51% of a program's branches, respectively. Fuzzer B has demonstrated a 1% improvement over Fuzzer A. However, suppose the additional branch Fuzzer B uncovers is an unlikely branch containing a rare bug. We argue that the slight increase in branch coverage does not convey the significance of uncovering this additional branch.

```
1  char *CUR;
2  int important = 1;
3  int a = 1;
4  #define CMP3(s, c1, c2, c3) \
5  ( ((unsigned char *) s)[0] == c1 && \
6    ((unsigned char *) s)[1] == c2 && \
7    ((unsigned char *) s)[2] == c3 )
8  int main (int argc, char **argv) {
9    CUR = argv[1];
10   if (CMP3(CUR, 'D', 'O', 'C')) {
11     CUR = CUR + 3;
12     parse_cmt();
13     if (parse_att()) {
14       important *= a; /* BUG */
15       ...
16     }
17   }
18   assert(important != 0);
19   return 0;
20 }
21 void parse_cmt() {
22   if (*CUR == '<' || *CUR == '>') {
23     a = 0;
24     CUR++;
25   }
26 }
27 int parse_att() {
28   if (CMP3(CUR, 'A', 'T', 'T'))
29     return 1;
30   return 0;
31 }
```

Fig. 2: Path-specific defect adapted from parser.c of `libxml`. Only one path causes the assertion on line 18 to fail.

In this paper, we propose accentuating the value of exploring a rare program path through a new coverage criterion: rare-path coverage or RP-Coverage. Intended to supplement existing coverage criteria, RP-Coverage quantifies a fuzzer's ability to explore unlikely program paths. RP-Coverage is a weighted form of path coverage in which each control-flow edge is weighed with the probability of that edge being taken, and the weight of a particular path is the product of its edges' weights. The paths with the lower probabilities are likely to significantly influence fuzzer efficacy and are more valuable to uncover.

To demonstrate the practical applicability of RP-Coverage, we have implemented and evaluated **R**are-**P**ath **P**robability Hound (RPP-HOUND), a tool for statically estimating program path probabilities. As a way to streamline this potentially cognitively taxing process, RPP-HOUND takes advantage of probabilistic logic programming as a means to declaratively encode the analysis rules. The rules' expressive and comprehensible nature makes them readily amenable to fine-tuning, auditing, and reuse. To evaluate RP-Coverage, we explore the RP-Coverage attained by state-of-the-art fuzzers and assess how RP-Coverage correlates with the fuzzer's ability to explore unlikely paths.

Our key finding is that RP-Coverage is more sensitive to rare paths than line or branch coverage. Specifically, we observe that RP-Coverage better reflects the value of uncovering rare paths with average increases up to 49.5%, 11.1%, and 15.4% for RP, line, and branch coverage, respectively. Additionally, we have shown the potential of RP-Coverage to identify fuzzing scenarios in which interesting rare paths remain unexplored. A fuzzer's RP-Coverage is noticeably smaller in these scenarios than line or branch coverage. The promise of RP-Coverage to supplement existing fuzzer fitness criteria is indicated by our experiments with state-of-the-art fuzzers, whose performance mostly converged across our benchmarks.

This paper makes the following contributions:

- It introduces RP-Coverage, a new criterion for measuring a program's rare execution path coverage.
- It concretely applies RP-Coverage to implement RPP-HOUND, a tool for estimating path probabilities; RPP-HOUND's design relies on probabilistic logic programming to express program analysis rules.
- It presents the findings of our empirical evaluation that applies RP-Coverage to state-of-the-art fuzzers to ascertain the utility of this coverage criterion to express a fuzzer's efficacy.

The rest of this paper is organized as follows. Section II presents a motivating example and describes the key elements of our approach; Section III discusses the design and implementation of RPP-HOUND; Section IV presents our evaluation results; Section V discusses the implication of our evaluation; Section VI compares our approach with the related state of the art; and Section VII presents concluding remarks.

II. MOTIVATION AND APPROACH

A key technical underpinning of RP-Coverage is a new treatment of path coverage and its calculation. Recall that existing notions of path coverage involve calculating the ratio of paths executed to total paths [5]. The traditional path coverage criterion assumes that all paths have equal weight. However, we observe that the probabilities of executing specific paths can vary widely. Driven by this observation, our insight is that path coverage should be calculated based on a weighted control flow graph (CFG) and that probability is a natural metric to represent the weights.

To motivate the need for a new type of coverage and its applicability to real-world scenarios, consider the C code snippet in Figure 2. This code snippet has been adapted from `libxml` and was used as a motivating example in [4]. We further adjust the snippet to introduce a critical post-condition invariant for the `important` global variable and an intentional bug that violates that invariant. Specifically, on line 14, the variable `important` is multiplied by variable `a` and updated, an assertion is added on line 18, and variable `a` is set to zero on line 23.

A violation of this invariant comprises a critical bug that must be detected before the software is released. However, modern fuzzers face challenges generating inputs that can trigger deep but serious bugs like this one. Notice that only three inputs would cause the control flow to pass through line 14. Furthermore, the control flow must pass through lines 23 and 14 in strict sequence for the assertion on line 18 to fail, triggering the bug. Consider the inputs "DOC>X" and

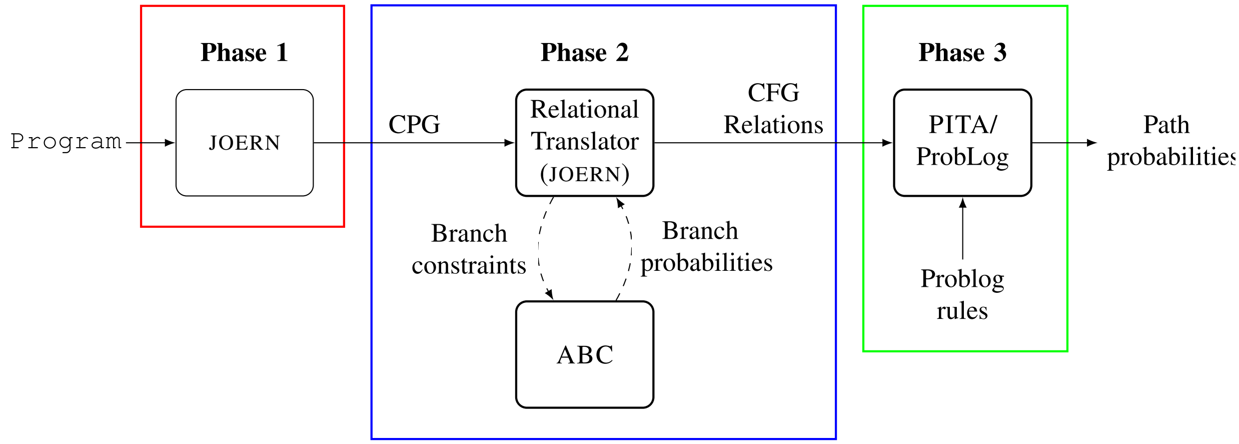


Fig. 3: RPP-HOUND system overview and data-flow diagram

"DOCATT". If a fuzzer generates both, the line coverage would include lines 14 and 23, albeit without triggering the bug. This scenario highlights a shortcoming of line and branch coverage criteria in quantifying how effective a fuzzer is in triggering this class of bugs. Path coverage would capture this scenario, but it treats identifying this specific path with the same value as any other.

We introduce RP-Coverage to express the influence of discovering new paths on program coverage. To that end, we assign each path a weight of $\frac{1}{P}$, where P is the path's probability. Thus, RP-Coverage is defined as the ratio of the total weights of covered paths to the total weights of all possible paths.

Our approach to calculating RP-Coverage starts from statically estimating the probability of a program path. We build on a technique presented in [4], which applies branch selectivity as a heuristic for estimating branch probabilities [6]. However, the approach implemented in RPP-HOUND simplifies the problem space and offers a declarative programming model. Specifically, the novelty of our technique lies in taking advantage of probabilistic logical programming as our analysis engine. Logic programming has been shown as a highly effective mechanism for performing scalable program analysis [7], [8]. Our technique uses probabilistic reasoning to concisely express path program paths and calculate their probabilities. Similarly to prior logic programming techniques for program analysis, we model the analyzed program's statements and structure as relational facts, but we also assign probabilities to the facts as appropriate. Then, we rely on the language engine to infer the overall path probabilities.

III. IMPLEMENTATION

We reify our approach as RPP-HOUND, depicted in Figure 3. RPP-HOUND comprises three distinct phases: (1) it converts a program into an intermediate analysis representation, (2) it translates the program into a database of probabilistic control-flow facts, and (3) it applies ProbLog rules to the facts to estimate probabilities for program paths. We detail each phase next.

A. Phase I: Generating Code Property Graph

A code property graph (CPG) represents a program by merging its abstract syntax tree (AST), CFG, and program dependency graph (PDG) [9]. The AST provides information on the program's source code; the CFG captures how the statements in the AST are connected; and the PDG represents the data dependencies within the statements. The power of CPGs lies in providing the advantage of all three graphs in a single convenient representation. While designed for concisely describing software vulnerabilities as graph traversals, CPGs provide a convenient intermediate representation for many program analysis problems.

RPP-HOUND's approach for calculating path probabilities involves constructing the program's probabilistic control flow graph. A CPG is an appropriate intermediate representation for approaching this task because it encodes control-flow relationships in the CFG and branch predicates in the PDG and AST. RPP-HOUND employs JOERN, an open-source CPG framework, for executing this phase.

B. Phase II: Translating CPGs into Prolog Facts

RPP-HOUND's second phase translates a given CPG into logical facts representing the program's probabilistic control flow, or probabilistic control flow graph (Prob-CFG) [4]. As its logic engine, RPP-HOUND uses ProbLog [10], a probabilistic extension of Prolog. The structure of the relational facts of ProbLog was amenable to concisely expressing Prob-CFGs. A separate Prob-CFG is generated for each program method. Following the commonly used terminology for specifying CPGs, we will use the term `method` to refer to both methods in object-oriented languages and functions in imperative languages. The logic required for generating a Prob-CFG is implemented in approximately 500 lines of Scala code. RPP-HOUND constructs a Prob-CFG for each method by labeling outgoing edges from control structure nodes (e.g., if, else, loops, etc.). The labels represent the probability that the control flow will take the given edge.

Probabilities are calculated with branch selectivity, as defined in Equation 1.

$$P(b) = \frac{|T_b|}{|D_b|},\ 0 \leq P(b) \leq 1. \tag{1}$$

The branch condition domains are identified by querying the control structure's data dependencies in the PDG and AST components of the CPG. The conditions and the domains of the variables that comprise them are represented as satisfiability modulo theory (SMT) constraints. The size of D_b and T_b are determined using the automata-based model counter (ABC) SMT solver [11]. This definition of probability assumes the variables in branch conditions are uniformly distributed.

Invoking an SMT solver is expensive. This approach is vulnerable to a performance bottleneck for programs with many branches or overly complex branch conditions. To mitigate the impact of this bottleneck, we further simplify the problem space. Instead of constructing SMT constraints representing a variable's actual domain—often as large as 2^{32}—we constrain the domains to 2^8. A statistical rule of thumb is that an event may be considered rare if its probability is less than 0.05 [12]. When implementing RPP-HOUND, we observed that many branch conditions produced probabilities less than 0.05 regardless of whether the domain size was 2^{32} or 2^8. In some conditions, this simplification over or under-approximates the branch's probability. We plan to evaluate the validity of this assumption more formally; however, we believe the performance gained from constraining the SMT problems outweighs the potential inaccuracies.

RPP-HOUND traverses the CPG for each method and outputs a set of ProbLog facts representing the program's probabilistic control flow. The output of this phase is the following control-flow relations:

- `cfg_edge(X, Y)`: a direct control flow edge between nodes `X` and `Y`.
- `P::cfg_edge(X, Y)`: a direct control flow edge between nodes `X` and node `Y` with a probability `P`.
- `branch(X, Y, TF)`: node `X` is a branch with an edge to `Y` when the condition evaluates to `TF` (1 for true or 0 for false).
- `loop(X)`: node `X` is the control structure for a loop.
- `method(X, Name)`: node `X` is the entry point for a method called `Name`.
- `calls(Caller, Callee, Call)`: `Caller` calls method `Callee` at node `Call`.
- `returns(Meth, Ret)`: `Meth` returns at node `Ret`.

This phase outputs a set of facts representing an *intra-procedural* Prob-CFG for each method. The declarative ProbLog rules infer the set of *inter-procedural* edges and context-sensitive paths, demonstrating another advantage of using declarative probabilistic programming for this analysis.

To achieve the necessary level of scalability, RPP-HOUND implements two optimizations that minimize the size of the Prob-CFG database. First, adjacent nodes with single incoming and outgoing control-flow edges are merged. Second, the analysis ignores any methods that are unreachable by user inputs. We manually specify functions that accept user input (e.g., `fread`, `scanf`, `getopt`, etc.) and then identify methods that are reachable by the return values of those functions. Both of these optimizations reduce the graph size and ProbLog memory overhead.

```
(1) icfg_edge(X,Y) :-
      cfg_edge(X,Y),
      \+calls(_,_,X).
(2) icfg_edge(X,Y) :- calls(_,Y,X).
(3) icfg_edge(X,Y) :-
      calls(_,M,Z),
      returns(M,X),
      cfg_edge(Z,Y).
```

Fig. 4: Inter-procedural Prob-CFG rules

C. Phase III: Estimating Path Probabilities

RPP-HOUND's third and final phase employs ProbLog to identify context-sensitive inter-procedural program paths and calculate their corresponding probabilities. Specifically, RPP-HOUND uses PITA [13], a ProbLog library for SWI-Prolog[14], a popular Prolog engine. We selected PITA because it fully supports the ProbLog syntax while simultaneously providing the mature functionality of SWI-Prolog.

Inter-procedural control edges are inferred from the intra-procedural control flow facts extracted in the previous phase. The rules in Figure 4 specify the relationship that infers an inter-procedural control-flow edge between nodes `X` and `Y`.

Rule (1) represents intra-procedural edges. Any existing CFG edges not originating from a call should be included in the inter-procedural CFG. Rule (2) represents edges between functions. There is an edge between nodes `X` and `Y` if `X` is a call site and `Y` is the callee. Note that `icfg_edge` facts are not inherently context-sensitive. Context sensitivity will be introduced when constructing execution paths. Rule (3) establishes a back-edge from a called function to its call site. There is an edge between nodes `X` and `Y` if `X` is the return site of a function that was called by the node immediately preceding `Y`.

The rules in Figure 4 are integrated into additional rules for traversing the graph and capturing the visited nodes. Notice that using the existing `icfg_edge(X, Y)` rules to create an `icfg_path(X, Y)` rule would fail to calculate the probability of two nodes being connected by a particular path. Instead, we used the rules that appear in Figure 5. These rules provide an expressive and comprehensible analysis specification for identifying particular execution paths.

Rules (1) and (2) indicate that the path has reached its endpoint—or reached the depth limit—and must be returned in the `Path` out variable. Rules (3-5) mirror the rules in Figure 4. Rules (4) and (5) also demonstrate how the analysis handles context sensitivity. The third term of `get_path` represents the calling context history. The head of the list represents the most recent call site. In rule (4), the analysis encounters a new call; the call node is pre-pended to the calling context

```
% Rule (1) Reached the end of the path
get_path(X,X,[Ctx|_],Visited,Path) :-
    reverse([(X,Ctx)|Visited],Path).

% Rule (2) Reached the depth limit
get_path(X,_,[Ctx|_],Visited,Path) :-
    \+db(less_than_depth(Visited)),
    reverse([(X,Ctx)|Visited],Path).

% Rule (3) Standard intra-procedural CFG edge
get_path(X,Z,[Ctx|Calls],Visited,Path) :-
    db(less_than_depth(Visited)),
    cfg_edge(X,Y),
    \+call_edge(X,Y),
    not(member((X,Ctx),T)),
    get_path(Y,Z,[Ctx|Calls],[(X,Ctx)|Visited],
        Path).

% Rule (4) Call edge; update the callsite context
get_path(X,Z,[Ctx|Calls],Visited,Path) :-
    db(less_than_depth(Visited)),
    calls(_,Y,X),
    not(member((X,Ctx),Visited)),
    get_path(Y,Z,[X,Ctx|Calls],[(X,Ctx)|T],Path).
```

```
% Rule (5) Call back-edge; update the callsite context
get_path(X,Z,[Ctx|Calls],Visited,Path) :-
    db(less_than_depth(Visited)),
    calls(_,Method,Ctx),
    returns(Method,X),
    cfg_edge(Ctx,Y),
    not(member((X,C),Visited)),
    get_path(Y,Z,Calls,[(X,Ctx)|Visited],Path).

% Rules (6) and (7) Loops are treated as single path
get_path(X,Z,[Ctx|Calls],Visited,Path) :-
    db(less_than_depth(Visited)),
    in_loop_cond(X,L),
    loop(L),
    branch(L,Y,1),
    not(member((L,Ctx),Visited)),
    get_path(Y,Z,[Ctx|Calls],[(L,Ctx)|Visited],Path).

get_path(X,Z,[Ctx|Calls],Visited,Path) :-
    db(less_than_depth(Visited)),
    in_loop_cond(X,L),
    loop(L),
    branch(L,Y,0),
    member((L,Ctx),T),
    get_path(Y,Z,[Ctx|Calls],[(L,Ctx)|Visited],Path).
```

Fig. 5: Inferring context-sensitive, inter-procedural program paths in ProbLog

list. In rule (5), the analysis encounters a method return and subsequently pops the current context off of the head of the list. As the analysis visits nodes, it is added to a visited list with its calling context. Rules (6) and (7) are included to prevent infinite loops. Our approach assumes that all loops will terminate and treats them as a single path instead of a branch. In Rule (6), if node X is a loop and that loop has not been visited, the path will always follow the true edge. In Rule (7), if node X is a loop that has already been visited, the next node will follow the false edge.

Using the Prolog findall predicate, RPP-HOUND finds all of the paths between the program's main method entry and exit nodes, capturing those paths with their probabilities. Each path is saved to a file that details the nodes in the path, with their associated source file and line number, and the path's aggregate probability. Similar to all path-based analyses, our approach must adequately manage the issue of path explosion. We limit the path depth to achieve a reasonable trade-off between resource consumption and obtaining meaningful results. As in [4], we selected 60 as the depth limit.

Currently, calculating RP-Coverage is an offline process. As a result, programs must be instrumented to output program execution traces. After RPP-HOUND has calculated path probabilities, we calculate RP-Coverage by ingesting the program traces and using them to identify what paths were executed. The probabilities of executed paths are used to calculate the paths' weights and overall RP-Coverage.

IV. EVALUATION

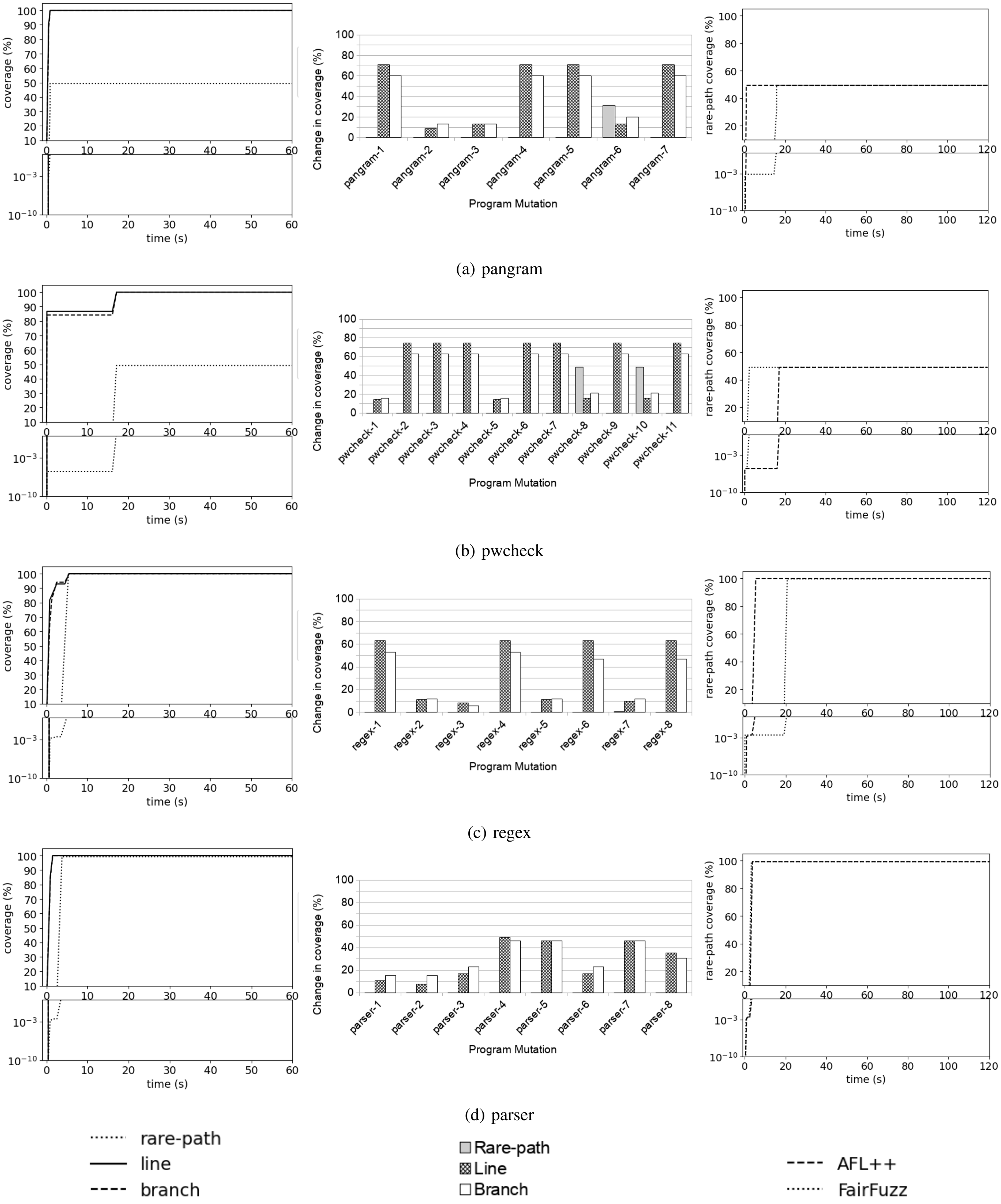
This work aims to answer the following research questions:

- **RQ1:** What is the utility of RP-Coverage as a metric for measuring fuzzer performance?
- **RQ2:** How does RP-Coverage correlate with existing coverage criteria for state-of-the-art fuzzers?
- **RQ3:** Does probabilistic logic programming offer viable machinery for constructing effective and maintainable rare-path analysis?

We evaluate the efficacy of RP-Coverage by applying it to four benchmark programs. To capture the program's execution trace dynamically, we instrument each program. In particular, we inject a function call at each point in the program that involves a change in control flow (e.g., branches, loops, and return statements). The injected function records the branch's line number and whether the control flow follows the branch's true or false edge. We calculate RP-Coverage by comparing the program's traces to its corresponding set of paths.

We introduce a post-condition invariant violation into our benchmark programs to study how program coverage and rare path execution correlate. We increment two global variables at the beginning of each basic block influenced by user input. For each pair of variables, we generate a mutation of the program that decrements one of the global variables. When the program has finished executing, we check whether the global variables are equal. If not, the path of interest has been executed, and we note that in the program trace. Each program mutation represents a mechanism for testing a distinct program path.

Our evaluation uses four benchmark programs: a regular expression program (regex) from an example in the KLEE project repository [15], a password checking program (pwcheck) from an example in the Symbolic PathFinder project [16], a program that checks if a string is a pangram (pangram) from a programming tutorial website [17], and a modified version of the program in Figure 2 (parser). Some program constructs make it difficult to match an execution



(a) pangram

(b) pwcheck

(c) regex

(d) parser

Fig. 6: Evaluation results; Column One: RP, line, and branch coverage achieved by executing AFL++ for 200,000 iterations; Column Two: Change in coverage when the fault is triggered in each benchmark mutation; Column Three: RP-Coverage achieved by executing AFL++ and FAIRFUZZ for 200,000 iterations

TABLE I: Cumulative RP, line, and branch coverage achieved when each fault is triggered

Fault #	Rare-path	Line	Branch
pangram			
1	8.0e-10	71.0	60.0
2	7.6e-4	87.0	80.0
3	4.8e-4	81.2	73.3
4	8.0e-10	71.0	60.0
5	8.0e-10	71.0	60.0
6	31.4	100.0	100.0
7	8.0e-10	71.0	60.0
pwcheck			
1	3.0e-6	86.7	84.2
2	3.0e-6	74.7	63.2
3	3.0e-6	74.7	63.2
4	3.0e-6	74.7	63.2
5	3.0e-6	86.7	84.2
6	3.0e-6	74.7	63.2
7	3.0e-6	74.7	63.2
8	49.1	100.0	100.0
9	3.0e-6	74.7	63.2
10	49.1	100.0	100.0
11	3.0e-6	74.7	63.2

Fault #	Rare-path	Line	Branch
regex			
1	2.3e-08	63.4	52.9
2	0.0030	93.0	94.1
3	0.0015	93.0	94.1
4	2.3e-08	63.4	52.9
5	4.7e-08	81.7	64.7
6	3.6e-13	63.4	47.1
7	99.9	100.0	100.0
8	3.6e-13	63.4	47.1
parser			
1	0.0015	86.1	84.6
2	0.0015	86.2	84.6
3	0.0030	100.0	100.0
4	0.0015	81.5	76.9
5	2.3e-08	78.5	76.9
6	0.0015	92.3	92.3
7	2.3e-08	78.5	76.9
8	9.1e-11	35.4	30.8
parser (path-based bug)			
9	99.2	100.0	100.0

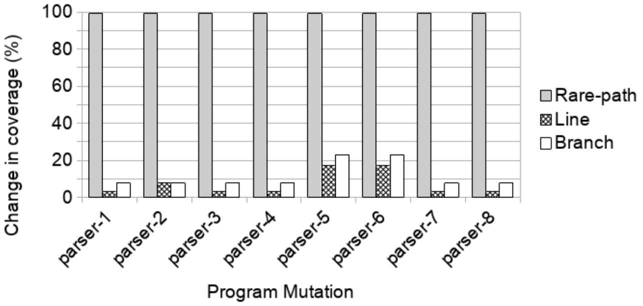


Fig. 7: Change in coverage when the path-based fault (fault #9) in parser is triggered

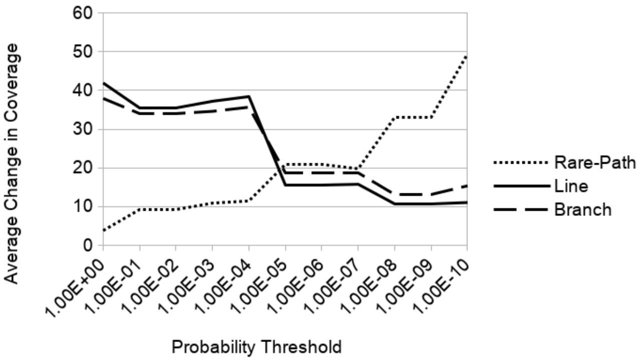


Fig. 8: Sensitivity analysis of the impact of coverage changes in response to variation in rarity thresholds

trace to its corresponding path, such as recursive function calls within a loop. We manually modify our subject programs to remove such troublesome constructs while preserving the program's original semantics as much as possible. In addition, we manage the input size for some programs to improve the likelihood that a fuzzer would maximize the number of execution paths it can take.

For this evaluation, we estimate the probabilities of all paths in the benchmark programs. These programs are small enough to identify all paths exhaustively—assuming all loops are treated as single iterations. In other words, we calculate the RP-Coverage by accounting for all paths.

We fuzz each original (unmutated) benchmark program with AFL++ [18] 200,000 times and each mutation 100,000 times. We track the line, branch, and RP coverage the fuzzer achieves over time for each program and mutation. The changes in coverage over time for the unmutated programs appear in the first column of Figure 6. Due to the nature of calculating RP-Coverage, if the program covers a common path, RP-Coverage increases only negligibly. As a result, at a given point, the cumulative RP-Coverage can potentially be smaller than 1%. The graphs in this column partition the results into two sub-graphs to represent these minimal changes. The top sub-graph presents the coverage percentages greater than 10% on a linear scale, and the bottom sub-graph presents the percentages less than 10% on a logarithmic scale.

Each program mutation can be interpreted as including a potential program fault. The purpose of these faults is to represent a particular program path. For each program fault, we identify the final line, branch, and RP coverage and the increase in coverage upon discovering the fault. Table I presents the final coverage metrics for each fault in the benchmark programs. The shaded rows in gray represent bugs that we consider rare. We define a rare bug as occurring along a path with a probability less than 1e-5. The changes in coverage at the time of each fault appear in the second column of Figure 6. A dedicated graph in Figure 7 depicts the changes in coverage for the path-based fault in parser detailed in Section II, as the only subject in which the fault depends on a specific statement execution order. Note that the same bug exists across all mutations of parser. The graph in Figure 7 presents results for the same bug evaluated multiple times.

To compare the cumulative RP-Coverage achieved by AFL++ and FAIRFUZZ, we fuzz each unmutated program for 200,000 iterations using FAIRFUZZ [3], with the results appearing in the third column of Figure 6.

To understand the correlation between path rarity and its impact on coverage metrics, we conduct a sensitivity analysis that calculates the average increases in line, branch, and RP coverage for different rarity thresholds. We present the average increases in coverage for thresholds ranging from 1% to 1e-10% in Figure 8.

To evaluate the benefits of RPP-HOUND's optimizations, we measure the number of ProbLog facts generated for each benchmark after enabling or disabling each optimization. Table II showcases the total number of facts, methods analyzed, control-flow edges, and method calls, as well as the average reduction achieved by applying the optimizations.

TABLE II: Impact of RPP-HOUND optimizations on ProbLog database size

Program and Opts	Facts	Methods	Edges	Calls
pangram-no-opts	247	11	148	24
pangram-no-reach	181	11	82	24
pangram-no-folding	168	2	115	1
pangram-optimized	98	2	45	1
pwcheck-no-opts	258	12	159	26
pwcheck-no-reach	184	12	85	26
pwcheck-no-folding	177	3	125	2
pwcheck-optimized	99	3	47	2
regex-no-opts	242	12	148	28
regex-no-reach	180	12	86	28
regex-no-folding	161	3	115	3
regex-optimized	94	3	48	3
parser-no-opts	208	14	134	20
parser-no-reach	148	14	74	20
parser-no-folding	134	5	99	4
parser-optimized	79	5	44	4
Average decrease	61.3%	74.0%	68.7%	89.3%

V. DISCUSSION

In this section, we discuss the results of our evaluation based on the research questions articulated in the previous section.

A. Utility of RP-Coverage

To answer **RQ1** and describe the utility of RP-Coverage, we take inspiration from economics, in which the term *elasticity* refers to how one variable responds to a change in another variable [19]. For example, if an increase in price for a commodity significantly lowers its demand, then demand is considered elastic or inelastic otherwise. By analogy, our evaluation confirms that RP-Coverage is indeed elastic to path probabilities. Discovering a path with a lower probability results in a greater increase in RP-Coverage than discovering one with a higher probability. In contrast, line and branch coverage lack that kind of elasticity. In all benchmarks, we observe a period during which a fuzzer achieves high levels of line and branch coverage (greater than 80%) but simultaneously achieves less than 1% RP-Coverage. This result confirms the elasticity of RP-Coverage and the lack thereof existing criteria.

Rare mutations are marked for each benchmark as follows: pangram: 2, 3, and 6; pwcheck: 8 and 10; regex: 2, 3, and 7; parser: 1, 3, 4, 6, and 9 (path-based). The corresponding graph shows that we observe one of the following two trends for most of these mutations. (1) RP-Coverage increases at a larger rate than line or branch; (2) the changes in line and branch coverage are relatively small compared to the less rare mutations. The path-based fault in the parser benchmark most strongly supports the second trend. When the fault is discovered, the increases in line and branch coverage range from 3-23%, yet the increase in RP-Coverage is over 99%. This large increase stems from the fuzzer discovering the specific fault-containing path. Inputs generated by the fuzzer previously covered the individual lines and branches along that path but were not in the strict order required to trigger the bug. The probability associated with this specific path is very low and thus greatly impacts the increase in coverage.

The results of the sensitivity analysis appearing in Figure 8 present further evidence of the elasticity of RP-Coverage. For higher probability thresholds, the line and branch coverage changes are high (30-40%), while RP-Coverage is low (less than 10%). Based on this analysis, a threshold of 1e-5 is the inflection point where the changes in all three coverage metrics are similar. At thresholds less than 1e-7, RP-Coverage starts growing much faster than line or branch. This dissimilarity in the coverage criteria for different probability thresholds indicates the usefulness of RP-Coverage in describing the behavior of fuzzers not currently covered by existing criteria. When combined with these existing criteria, RP-Coverage can provide a more complete picture of a fuzzer's effectiveness.

We observe that line and branch coverages adequately measure a fuzzer's ability to discover bugs on common paths. However, discovering bugs that manifest along a rare path produces a smaller increase in these coverages than RP-Coverage. We conclude that a large increase in RP-Coverage suggests that the fuzzer is more adequately exploring the program's less likely—and potentially more interesting—paths.

B. RP-Coverage Performance and Applicability

Our evaluation suggests that RP-Coverage has the potential to provide additional insights into the performance of fuzzers. To answer **RQ2**, we aim to understand the ability of RP-Coverage to reveal the peculiarities of fuzzer performance not identified by existing metrics. In all benchmarks, we observe that the achieved percentage of RP-Coverage is strictly less than line or branch. While regex and parser achieve high levels of RP-Coverage, pangram and pwcheck stagnate at approximately 50%. Additionally, in our pwcheck benchmark, we observe noticeable increases in RP-Coverage thousands of iterations after seeing a similar increase in line or branch.

Furthermore, we observe similar RP-Coverage trends for each benchmark when utilizing different fuzzers. The rate at which the RP-Coverage is achieved varies slightly. However, in all four benchmarks, the final level of RP-Coverage achieved converges. This convergence suggests that RP-Coverage has promise to be a consistent metric across multiple fuzzers. Contrary to our intuition, FAIRFUZZ's RP-Coverage increases exhibited neither a faster rate nor a larger amount. Despite FAIRFUZZ's design to bias towards rare paths, our experiments show no noticeable performance improvements. In contrast, its coverage increases slower than AFL++ in three benchmarks. One explanation is the peculiarity of FAIRFUZZ's allocation of computational cycles to mutate inputs selectively, as this strategy would impact the number of iterations it can complete per second. The observed divergences in RP-Coverage and existing criteria suggest that the former can reveal additional insights about fuzzer performance.

C. Rare Path Analysis with ProbLog

Although seemingly of only engineering significance, the design and implementation of RPP-HOUND answers **RQ3** by offering valuable insights for related efforts in creating advanced program analysis infrastructures. While traditional approaches for fuzzing rare paths require a serious effort to understand and modify mature program analysis infrastructures, our experiences show that the same functionality can be provided as declarative rules that mature probabilistic engines can efficiently execute. Logic languages require a database of facts, whose construction in RPP-HOUND is accomplished via the power of functional programming techniques of Scala, with the entire implementation comprising less than 500 LOC of Scala and seven ProbLog rules.

A naive implementation of our approach would not achieve the desired performance. We quantify the value of our optimizations in Table II. We observe that merging adjacent nodes largely impacts the total number of facts and control flow edges. Incorporating user input reachability similarly impacts the total number of methods analyzed. We gain the benefits of both optimizations by applying them in tandem. Across our benchmarks, we observe that applying both optimizations decreases the number of facts, methods, edges, and calls by 61.3%, 74%, 68.7%, and 89.3%, respectively. These results highlight the importance of properly optimizing even the most declarative implementation to achieve the desired performance. Based on these observations and our experiences, we conclude that probabilistic logic programming provides a promising mechanism for specifying and rare paths.

D. Limitations and Threats to Validity

As a proof-of-concept, RPP-HOUND and its implementation are subject to several limitations. First, the analysis only explores paths to a depth of 60 nodes. This depth limit may cause the analysis to overestimate path probabilities. However, we selected the depth limit demonstrated as effective by existing literature [4].

To estimate the probability of control-flow paths, we rely on a heuristic applied successfully in prior rare path analyses [6], [4]. Nevertheless, this heuristic, branch selectivity, assumes the values assigned to variables are distributed uniformly. Although this condition rarely holds in practice, the simplicity of the heuristic allows for efficient implementation. Despite the chance for misestimated probabilities, this limitation is an artifact of our implementation, not the RP-Coverage concept.

Our evaluation is subject to both internal and external validity threats, which we outline next. Our findings depend on the correctness of RPP-HOUND's probability calculations. In lieu of established correctness benchmarks, we had to rely on applying our testing discipline to check our implementation logic and its performance in the field. To further mitigate this threat, we manually check the calculated probabilities at the branch level, relying on the maturity of ProbLog inferencing to correctly calculate probabilities at the path level.

As is commonly the case, our selection of evaluation subjects is subject to bias. To mitigate this bias, we limited our selection exclusively to third-party programs, which we only adapted to meet the specific objectives of our experiments.

Some facets of our evaluation depend on artificially injected faults that are always introduced at the basic block level. Had the faults been introduced elsewhere, our evaluation findings might differ. However, following the basic block-level strategy reduces the bias inherent in determining where to inject faults.

In determining whether a mutation path is rare, we apply the probability threshold 1e-5. Although this particular threshold may seem arbitrary, its selection relies on sensitivity analysis presented in Figure 8.

Due to the number and size of our evaluation subjects, our findings may not generalize to a real-world code base, posing an external threat to validity. Only future work can help determine how serious this threat is. However, we focus on achieving manageable analysis workloads when selecting our subjects.

E. Future Directions

In addition to answering the three research questions above, our evaluation suggests that RP-Coverage can be a starting point for several future lines of inquiry. In conveying program coverage, combining coverage criteria generally proves more successful than when taken in isolation [20]. An interesting future research effort may integrate RP-Coverage into a fuzzer to study RP-Coverage's ability to guide fuzzing more effectively. Further, RP-Coverage can provide value by supplementing other domains that use program coverage metrics. For example, evaluating the comprehensiveness of test suites or generating automated test cases.

The analysis techniques presented herein are not specific to any particular language. Our design leverages CPGs as an intermediate representation and can potentially be applied to other language environments or multilingual programs simply by supplying small language-specific bindings.

ProbLog's performance as an inference engine proves sufficient for our subject applications. If scalability and performance become an issue in subsequent efforts, one may explore the applicability of probabilistic dialects of Datalog, a logic language designed explicitly for efficiently handling large volumes of data.

VI. RELATED WORK

RP-Coverage and RPP-HOUND are related to several areas of the related state of the art. Next, we describe the most closely related prior works.

Guided fuzzing and fuzzing coverage metrics: Several prior efforts demonstrate the benefits of biasing fuzzers toward rare program paths. FAIRFUZZ [3] dynamically updates a path's rarity based on the frequency of the fuzzer hitting each branch. In contrast, [4] introduces a static heuristic based on branch selectivity [6]. Our approach takes inspiration from [4] but differs in its implementation strategy. In addition to proposing a new coverage criterion, our system design streamlines the analysis by offloading complicated calculations to a probabilistic logic engine.

Additional works have studied methods for improving path coverage. PathAFL [21] uses h-paths to assist fuzzers in identifying interesting paths to explore further. CollAFL [22] introduces a coverage-sensitive fuzzer and new seed selection strategies that increase overall path coverage. Our work takes a different focus by supplementing these techniques with an additional metric to assess their performances.

Combining coverage criteria is stronger at detecting faults than any individual criterion [20]. As a result, prior work has explored introducing new coverage metrics to fill in the gaps left by existing ones. DeepXplore [23] introduces neuron coverage, which addresses the shortcomings of traditional coverage metrics when analyzing deep learning systems. [24] introduces program state coverage. Although analogous to these prior works in proposing a new coverage criterion, the uniqueness of our approach lies in its treatment of path weight.

Declarative program analysis: This work draws on the extensive prior research that applies logic languages to various problems in program analysis. Extensive literature demonstrates the efficacy of using such languages for declaratively specifying sophisticated analyses for their highly efficient execution [25], [26]. Specific applications range from low-level analyses, such as points-to analysis [7], [27], [8], to higher-level problems, like identifying structural program dependencies [28] or code property violations [29]. Numerous frameworks use logic languages to specify analyses, including Doop [8], Petablox [30], and souffle [31]. Our approach also benefits from the power of logic programming to express complex program analysis logic concisely, but we differ in the logic language we use. Given the increasing applicability of probabilistic reasoning in various software engineering problems [32], our approach opens up promising opportunities for expressing the solutions to these problems using probabilistic logic languages.

VII. Conclusions

We have presented Rare-Path Coverage, or RP-Coverage for short, a new program coverage metric that accounts for the probabilities of control-flow paths. We have demonstrated the potential of RP-Coverage as a viable means of evaluating fuzzing efficacy and performance. Based on our findings, we proposed RP-Coverage to supplement existing coverages to provide developers with a more complete picture.

Our empirical evaluation demonstrates that RP-Coverage is sensitive, or elastic, to path probabilities. As such, it indicates a fuzzer's ability to discover less likely but potentially more interesting portions of the program. Based on our experiences and evaluation, we anticipate that RP-Coverage can provide meaningful insights for any software engineering domain that uses program coverage as a performance metric.

We have concretely demonstrated how RP-Coverage can be implemented using cutting-edge program analysis approaches. In particular, the novel aspect of our proof-of-concept implementation—**R**are-**P**ath **P**robability Hound—is applying probabilistic logic programming for expressing program representation and analysis rules. Our experiences indicate that probabilistic logic programming offers an elegant approach to solving other software engineering problems. RPP-Hound's source code is available at https://github.com/SoftwareInnovationsLab/rpp-hound.

Acknowledgments

The authors thank the anonymous reviewers, whose insightful comments helped improve the quality of this paper. This research is supported by NSF grants #2232565 and #2106420.

References

[1] ISO/IEC/IEEE, "International Standard - Software engineering - Software life cycle processes - Maintenance," *ISO/IEC/IEEE 14764:2022(E)*, pp. 1–46, 2022.

[2] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22. New York, NY, USA: ACM, 2022, pp. 1621–1633.

[3] C. Lemieux and K. Sen, "FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2018, pp. 475–485.

[4] S. Saha, L. Sarker, M. Shafiuzzaman, C. Shou, A. Li, G. Sankaran, and T. Bultan, "Rare path guided fuzzing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2023, pp. 1295–1306.

[5] R. Bierig, S. Brown, E. Galván, and J. Timoney, *Essentials of Software Testing*. Cambridge University Press, 2021.

[6] S. Saha, M. Downing, T. Brennan, and T. Bultan, "PReach: A heuristic for probabilistic reachability to identify hard to reach statements," in *Proceedings of the 44th International Conference on Software Engineering*. New York, NY, USA: ACM, 2022, pp. 1706–1717.

[7] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2004, pp. 131–144.

[8] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. New York, NY, USA: ACM, 2009, pp. 243–262.

[9] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.

[10] L. De Raedt, A. Kimmig, and H. Toivonen, "ProbLog: A probabilistic Prolog and its application in link discovery," in *Proceedings of the 20th International Joint Conference on Artificial Intelligence*. San Francisco, CA, USA: Morgan Kaufmann Publishers, 2007, pp. 2462–2467.

[11] A. Aydin, L. Bang, and T. Bultan, "Automata-based model counting for string constraints," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds. Cham: Springer International Publishing, 2015, pp. 255–272.

[12] S. Tenny and I. Abdelgawad, *Statistical significance*. StatPearls Publishing, 2023.

[13] F. Riguzzi and T. Swift, "The pita system: Tabling and answer subsumption for reasoning under uncertainty," *Theory and Practice of Logic Programming*, vol. 11, no. 4-5, pp. 433–449, 2011.

[14] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[15] KLEE, "regex," https://github.com/klee/klee/tree/master/examples/regexp.

[16] jpf-symbc, "PassCheck," https://github.com/SymbolicPathFinder/jpf-symbc/blob/master/src/examples/strings/PassCheck.java.

[17] GeeksforGeeks, "C program to check if string is pangram," https://www.geeksforgeeks.org/c-program-to-check-if-string-is-pangram/.

[18] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[19] S. A. Greenlaw, D. Shapiro, and D. MacDonald, *Principles of Economics 3e*. OpenStax, 2022. [Online]. Available: https://openstax.org/books/principles-economics-3e/pages/1-introduction

[20] H. Hemmati, "How effective are code coverage criteria?" in *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 2015, pp. 151–156.

[21] S. Yan, C. Wu, H. Li, W. Shao, and C. Jia, "PathAFL: Path-coverage assisted fuzzing," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2020, pp. 598–609.

[22] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy*. IEEE, 2018, pp. 679–696.

[23] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, 2017, pp. 1–18.

[24] K. E. Someoliayi, S. Jalali, M. Mahdieh, and S.-H. Mirian-Hosseinabadi, "Program state coverage: A test coverage metric based on executed program states," in *IEEE 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2019, pp. 584–588.

[25] S. Dawson, C. R. Ramakrishnan, and D. S. Warren, "Practical program analysis using general purpose logic programming systems—a case study," in *ACM SIGPLAN Notices*, vol. 31. New York, NY, USA: ACM, 1996, pp. 117–126.

[26] S. S. Huang, T. J. Green, and B. T. Loo, "Datalog and emerging applications: an interactive tutorial," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2011, pp. 1213–1216.

[27] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, "Context-sensitive program analysis as database queries," in *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. New York, NY, USA: ACM, 2005, pp. 1–12.

[28] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and continuous checking of structural program dependencies," in *Proceedings of the 30th International Conference on Software Engineering*. New York, NY, USA: ACM, 2008, pp. 391–400.

[29] L. St. Amour, "Interactive synthesis of code-level security rules," Master's thesis, Northeastern University Boston, 2017.

[30] M. Naik, "Petablox: Large-scale software analysis and analytics using Datalog," *Tech. Rep.*, 2020.

[31] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *Computer Aided Verification*. Cham: Springer International Publishing, 2016, pp. 422–430.

[32] L. St. Amour and E. Tilevich, "Toward declarative auditing of java software for graceful exception handling," in *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. New York, NY, USA: ACM, 2024, pp. 90–97.