PReach: A Heuristic for Probabilistic Reachability to Identify Hard to Reach Statements *

Seemanta Saha
University of California, Santa Barbara
Santa Barbara, CA, USA
seemantasaha@cs.ucsb.edu

Mara Downing
University of California, Santa Barbara
Santa Barbara, CA, USA
maradowning@cs.ucsb.edu

Tegan Brennan
University of California, Santa Barbara
Santa Barbara, CA, USA
tegan@cs.ucsb.edu

Tevfik Bultan
University of California, Santa Barbara
Santa Barbara, CA, USA
bultan@cs.ucsb.edu

ABSTRACT

We present a heuristic for approximating the likelihood of reaching a given program statement using 1) branch selectivity (representing the percentage of values that satisfy a branch condition), which we compute using model counting, 2) dependency analysis, which we use to identify input-dependent branch conditions that influence statement reachability, 3) abstract interpretation, which we use to identify the set of values that reach a branch condition, and 4) a discrete-time Markov chain model, which we construct to capture the control flow structure of the program together with the selectivity of each branch. Our experiments indicate that our heuristic-based probabilistic reachability analysis tool PReach can identify *hard to reach* statements with high precision and accuracy in benchmarks from software verification and testing competitions, Apache Commons Lang, and the DARPA STAC program. We provide a detailed comparison with probabilistic symbolic execution and statistical symbolic execution for the purpose of identifying hard to reach statements. PReach achieves comparable precision and accuracy to both probabilistic and statistical symbolic execution for bounded execution depth and better precision and accuracy when execution depth is unbounded and the number of program paths grows exponentially. Moreover, PReach is more scalable than both probabilistic and statistical symbolic execution.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Software verification**; **Automated static analysis**; • **Theory of computation → Program analysis**.

*This material is based on research sponsored by NSF under grants CCF-2008660, CCF-1901098 and CCF-1817242. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05.
https://doi.org/10.1145/3510003.3510227

ACM Reference Format:
Seemanta Saha, Mara Downing, Tegan Brennan, and Tevfik Bultan. 2022. PReach: A Heuristic for Probabilistic Reachability to Identify Hard to Reach Statements . In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510227

1 INTRODUCTION

Software quality assurance is one of the most fundamental problems in computing. The most common software quality assurance technique is software testing. Although there has been a surge of progress in automated software testing techniques such as random testing, fuzzing and symbolic execution in recent years, there are remaining challenges. On one hand, fuzzing and random testing techniques are comparatively scalable, but have difficulty in exploring *hard to reach* program paths. On the other hand, symbolic execution based techniques can explore *hard to reach* program paths by solving path constraints, but are not as scalable.

Hybrid testing techniques [16, 26, 38, 39, 42] combine concrete (e.g., random testing, fuzzing) and symbolic techniques in order to improve testing effectiveness. Typically, a strategy function for hybrid testing decides when to apply concrete techniques and when to apply symbolic techniques to achieve scalable and effective exploration of the program behaviors. In order to choose between concrete and symbolic approaches, most existing strategies assess the difficulty of concrete testing based on the saturation of random testing [26, 38] or probabilistic program analysis [39, 42]. Determining the likelihood (or, conversely, difficulty) of reaching a program statement is critical for assessing the difficulty of concrete testing, and hence developing an effective hybrid testing strategy. There are two existing approaches that address this problem: probabilistic and statistical symbolic execution.

Probabilistic symbolic execution [20] is an extension of symbolic execution that computes probabilities of program paths. However, probabilistic symbolic execution suffers from the same limitations as symbolic execution: 1) It can only analyze program behaviors up to a certain fixed execution depth, hence it cannot analyze behaviors of arbitrarily large program paths. 2) Due to exponential increase in number of paths with increasing execution depth (path explosion problem), the cost of symbolic execution increases exponentially with increasing execution depth. 3) Although the sizes of path constraints generated by symbolic execution increase linearly with the execution depth, since the worst case complexity of constraint

solvers is exponential, the linear increase in path constraint sizes can lead to exponential increase in analysis cost. Hence, path explosion combined with increasing sizes of path constraints can lead to double exponential blow up in the cost of symbolic execution, limiting its practical applicability.

Statistical symbolic execution [18] is more efficient and scalable compared to probabilistic symbolic execution [20]. However, it cannot compute precise reachability probabilities, rather provides approximate reachability probabilities with statistical guarantee. Statistical symbolic execution suffers from similar issues as probabilistic symbolic execution. There are two variants of statistical symbolic execution: 1) statistical analysis based on Monte Carlo sampling of symbolic paths, and 2) hybrid analysis combining both statistical and exact analysis based on informed sampling. One of the drawbacks of pure statistical sampling is that it needs to sample a large number of paths to achieve high statistical confidence. Informed sampling obtains more precise results and converges faster than a purely statistical analysis, but its effectiveness suffers when the number of program paths grows exponentially.

In this paper, we present a heuristic for probabilistic reachability analysis to identify *hard to reach* program statements that addresses the above shortcomings of probabilistic symbolic execution and statistical symbolic execution. In particular, 1) our approach can model behaviors of arbitrarily long paths, 2) it does not suffer from path explosion, i.e., the cost of our analysis increases polynomially with the size of the program (and does not depend on the execution depth) [23], and finally, 3) it solves constraints arising from branch conditions rather than path constraints which reduces the cost of constraint solving.

Our approach, which we implemented in our tool PReach, works as follows (Figure 1). In order to compute reachability probability of statements, we introduce a concept called *branch selectivity* that determines the proportion of values satisfying a given branch condition. A branch is very selective if only a few values satisfy the branch condition. On the other hand, if a lot of values satisfy the branch condition, then the branch is not very selective. Given a target statement in a program, PReach identifies the input dependent branch conditions that influence the reachability probability of that statement using dependency analysis. Then, PReach constructs a discrete-time Markov chain model from the control flow graph of the program by computing branch selectivity of each branch condition that influences the reachability probability of the target statement. PReach uses abstract interpretation to determine the set of values that reach each branch condition and model counting to compute the branch selectivity value for each branch in the program that influences statement reachability. Finally, PReach uses a probabilistic model checker to compute the reachability probability of the target statement based on the constructed discrete-time Markov chain model.

One shortcoming of our approach is that it is not a sound program analysis technique and hence, it does not provide guarantees in terms of the precision or accuracy of the reachability probabilities it reports. On the other hand, though, bounded symbolic execution is theoretically sound up to the execution bound, and probabilistic symbolic execution can quantify how much of the execution space is not explored due to the execution bound [18], for unbounded executions, both probabilistic symbolic execution and statistical symbolic execution are not sound either.

We experimentally evaluate PReach on programs from the SV-COMP benchmark set used in Competition on Software Verification [8] and Competition on Software Testing [9]. Each program in this benchmark set contains an assert statement. We use these assert statements as the target of our probabilistc reachability analysis. We evaluate the effectiveness of our technique in separating *hard to reach* assert statements (i.e., assert statements with low reachability probability) from easy to reach assert statements (i.e., assert statements with high reachability probability) using a probability threshold (i.e., if the reachability probability of a statement is below the given threshold we classify it as *hard to reach*).

In order to determine the ground truth, we use a generator based random fuzzer that is based on JQF [28] and ZEST [29]. We set a time limit for the random fuzzer, and the assert statements that are not reached within the given timeout are marked as the *hard to reach* assert statements. Of the 142 programs we used in our experiments, the random fuzzer times out on 51 programs. PReach classifies the programs that the random fuzzer times out on as hard-to-reach, with 95.8% precision and 95.1% accuracy. In particular, our technique correctly classifies 135 out of 142 programs and generates only 2 false positives (reports *hard to reach* although the fuzzer does not time out) and 5 false negatives (reports *easy to reach* although the fuzzer times out).

In order to further evaluate the effectiveness of our probabilistic reachability analysis, we provide a detailed experimental comparison with the probabilistic symbolic execution (PSE) [20] and statistical symbolic execution (SSE) [18] extensions to Symbolic PathFinder (SPF) [30] tool. Experimental results show that for programs with bounded execution depth, PSE achieves very high precision and accuracy to identify *hard to reach* cases. However, PReach outperforms PSE for programs with unbounded execution depth in terms of precision, accuracy and average analysis time. For large search depths PSE is unable to analyze 38% of the target programs demonstrating its limitations in terms of applicability and scalability, whereas PReach can analyze 100%. We compare PReach with SSE on the set of programs that PSE performs poorly. SSE was unable to analyze 27% of these programs and PReach outperforms SSE in terms of precision, accuracy, and average analysis time.

Finally, we analyze 24 target statements in 18 methods from Apache Commons Lang [1] and DARPA STAC Benchmarks [4]. PReach can classify 19 of the 24 target statements correctly demonstrating its effectiveness on real world programs, whereas PSE and SSE were able to successfully analyze and classify only one.

2 OVERVIEW

We formalize probabilistic reachability analysis as follows. Given a program p, let i denote the input for the program, and I denote the domain of inputs (i.e., $i \in I$). Note that i can be a scalar value, a tuple, or a list of values. Given a target statement t in program p, the goal of probabilistic reachability analysis is to determine how likely it is to reach target statement t. We do this by determining how likely it would be to pick inputs that result in an execution that reaches t. In order to determine how likely it would be to pick

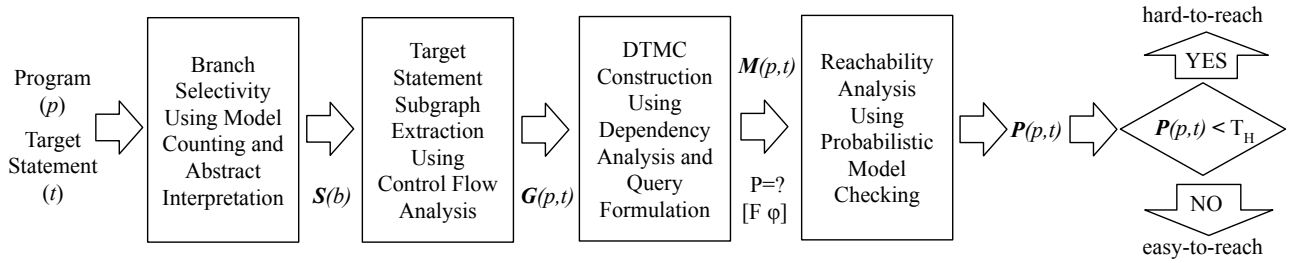


Figure 1: Probabilistic Reachability Heuristic in PREACH

```
1  public class Main {
2    public static void main(String[] args) {
3      int arg = Verifier.nondetInt();
4      if ( arg < 0 )
5        return;
6      int x = arg / 5;
7      int y = arg / 5;
8      Main inst = new Main();
9      inst.test(x, y);
10   }
11   public void test(int x, int z) {
12     System.out.println("Testing ExSymExe7");
13     int y = 3;
14     z = x - y - 4;
15     if ( z != 0 )
16       System.out.println("branch FOO1");
17     else {
18       System.out.println("branch FOO2");
19       assert false;
20     }
21     if ( y != 0 )
22       System.out.println("branch BOO1");
23     else
24       System.out.println("branch BOO2");
25   }
26 }
```

Figure 2: An example based on SV-COMP benchmark

such inputs, we determine the probability of picking such inputs if inputs are chosen randomly. We define $\mathcal{P}(p,t)$ as:

> $\mathcal{P}(p,t)$ denotes the probability of reaching statement s during the execution of program p on input i if i is selected randomly from the input domain I.

We assume uniform distribution of inputs in our current implementation. However, our technique can be easily extended to support any input distribution by integrating usage profiles [18] used in other probabilistic analysis techniques.

It is well-known that determining reachability of a statement in a program is an uncomputable problem. Hence, determining $\mathcal{P}(p,t)$ precisely is also an uncomputable problem. In this paper we present a heuristic approach that approximates $\mathcal{P}(p,t)$. We report the reachability probability as a real number between 0 and 1.

Branch Selectivity. Our heuristic approximation of $\mathcal{P}(p,t)$ relies on a concept we call *branch selectivity*. Given a branch b, branch selectivity $\mathcal{S}(b)$ is proportional to the ratio of the number of values that satisfy the condition for branch b to the total number of values in the domain of condition for branch b. Formally, given a branch b, let D_b denote the Cartesian product of the domains of the variables that appear in b, and let $T_b \subseteq D_b$ denote the set of values for which branch b evaluates to true. Let $|D_b|$ and $|T_b|$ denote the number of elements in these sets, respectively. Then, $\mathcal{S}(b) = \frac{|T_b|}{|D_b|}$.

So, the selectivity of a branch gets closer to 0 as the number of values that satisfy the branch condition decreases, and it gets closer to 1 as the number of values that satisfy the branch condition increases. If we think of branch as a sieve, when $\mathcal{S}(b) = 0$ branch b does not allow any value to pass, and when $\mathcal{S}(b) = 1$ branch b allows all values to pass. Note that, if we pick values from the domain D randomly with a uniform distribution, then $|T_b|/|D_b|$ corresponds to the probability of picking a value that satisfies the branch condition. The branch becomes more selective as the probability of picking a value decreases.

An Example. Consider the integer-manipulating program in Figure 2. This program is a modified version of an example from the *jpf-regression* directory of the SV-COMP benchmark used for software verification and testing competitions [8]. The target statement is the assertion statement in line 19. The *arg* variable's value is a randomly generated integer value and it denotes the input to this program. The question we want to answer for this program is: *how likely it is to reach the assertion statement at line 19 if we randomly generate values for the arg variable?*

The first conditional statement at line 4 ignores all the negative values. At line 15, possible values for z can be any randomly generated positive value, divided by 5, minus 7. Now, the assertion at line 19 is reachable when value of z is equal to 0. The likelihood of the value of z being equal to 0 is low if the input is a random number generated from a uniform distribution. Therefore, the probability of reaching the assert statement in this program is low.

Our analysis uses branch selectivity based on model counting to successfully determine the reachability probability of the assert statement in this program. We inspect each branch condition leading to the assertion to determine how selective the branch is (i.e. what ratio of input values satisfy the branch). If we assume a domain of integer values, then for the conditional statement $arg < 0$, branch selectivity is calculated as half of the domain. Therefore, the possible values reaching the assertion is reduced to half. Next, for the next conditional statement, $z \neq 0$, branch selectivity is close to 1. Most values satisfy this constraint and conversely, only 1 value of z satisfies its negation. The assertion lies on the else branch of this condition, making it reachable only for one value of z.

Using the branch selectivity values computed at these branches, we convert the control flow graph of the program to a discrete time Markov chain as shown in Figure 4c. We use a probabilistic model checker to analyze the Markov chain and obtain a probabilistic

measure for assertion reachability. For the running example, this reachability probability is computed as $0.5 \times (2.32e^{-10})$. The value 0.5 arises from the branch selectivity for the branch condition $arg < 0$ and $2.32e^{-10}$ arises from the branch selectivity for the branch condition $z \neq 0$. The reachability probability of the assertion statement is then reported as $1.16e^{-10}$, hence this statement would be classified as a *hard to reach* statement by our analysis since it has a low reachability probability.

To assess the success of our analysis for this example, we run a generator based random fuzzer with a timeout of 1 hour. We find that the fuzzer cannot generate an input to reach the assertion. The fuzzer generates 4,103,625 inputs and none of them reach the assertion, which supports the result of our analysis.

Since our analysis does not precisely represent the original semantics of the program, we cannot make soundness claims about the probability computed by our heuristic. In general case, our analysis may over or under approximate the reachability probability. By integrating abstract interpretation techniques to our analysis, we achieve better precision which we will discuss in section 3.2.

In the following sections we discuss how we compute and use branch selectivity values together with control flow, dependency analysis and abstract interpretation to extract a discrete-time Markov chain and then use probabilistic model checking to compute approximations of reachability probability.

3 A PROBABILISTIC REACHABILITY HEURISTIC

We approximate $\mathcal{P}(p,t)$ using a combination of control flow, dependency analysis, abstract interpretation, model-counting and probabilistic model checking. First, we discuss how model counting constraint solvers and abstract domains can be used to compute branch selectivity. Then, we use control flow and dependency analysis and branch selectivity to transform the program's control flow graph into a Markov chain. We form queries on this Markov chain solvable by probabilistic model checking whose solutions approximate $\mathcal{P}(p,t)$. If $\mathcal{P}(p,t)$ is less than a given threshold T_H, target statement is predicted as *hard to reach*. We discuss these steps below.

3.1 Branch Selectivity

The enabling technology for computing branch selectivity is model counting. Model counting is the problem of determining the number of satisfying solutions to a set of constraints. A model counting constraint solver is a tool which, given a constraint and a bound, returns the number of satisfying solutions to the constraint within the bound. For a branch condition b, recall that $\mathcal{S}(b) = \frac{|T_b|}{|D_b|}$, where D_b is the Cartesian product of the domains of the variables that appear in b and T_b is the set of values in D_b for which b evaluates to true. For a given b and D_b, a model-counting constraint solver computes $|T_b|$. Then, using $|T_b|$ we compute $\mathcal{S}(b)$.

We use the Automata-Based Model Counter (ABC) tool, which is a constraint solver for string and numeric constraints with model counting capabilities [2]. The constraint language for ABC supports linear arithmetic constraints as well as typical string operations. In order to compute $\mathcal{S}(b)$ we first extract the branch condition from the program and then generate a formula in the SMT-LIB format that corresponds to the branch condition. Then, we send the formula to ABC as model counting query.

```
1  public void test(int x) {
2    if(x >= 0) {
3      int y = -x;
4      if (y > 0) {
5        assert false;
6       }
7    }
8  }
```

(a) Using interval analysis

```
1  public void test(int x,
2    int z, int r) {
3    int y = 3;
4    r = x + z;
5    z = x - y - 4;
6    if (x < z)
7      assert false;
8  }
```

(b) Using relational analysis

Figure 3: Refined branch selectivity

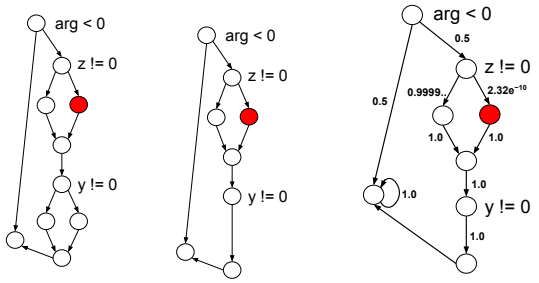
3.2 Refined Branch Selectivity

Abstract interpretation techniques overapproximate program behaviors by interpreting programs over abstract domains. Our key insight here is that it is possible to use abstract interpretation to refine and restrict the set of values that variables can take at each branch in order to better approximate the branch selectivity. Given a branch b, using abstract interpretation we generate a refinement condition R_b to overapproximate the set of values that the variables can take at that branch. R_b is then conjoined with T_b and D_b to compute refined branch selectivity $\mathcal{RS}(b)$. For a branch condition b, refined branch selectivity is defined as $\mathcal{RS}(b) = \frac{|T_b \wedge R_b|}{|D_b \wedge R_b|}$.

To implement the refined branch selectivity, we use state-of-the-art Java numeric analysis tool JANA [41] which supports two different abstract domains, intervals [21] and polyhedra [37], where polyhedra domain leads to more precise results however it is less scalable. We experimented with both of these domains to extract the refinement conditions R_b for each branch using interval analysis and relational (using polyhedra domain) analysis. We call these implementations PReach-I and PReach-P, respectively.

Consider the two code snippets from Fig. 3a and 3b. At line 4 in Fig. 3a, T_b and D_b are $y > 0$ and *True* respectively. $\mathcal{S}(b)$ computed by PReach is 0.25 predicting incorrectly that the assertion is reachable. Applying either interval or relational analysis, R_b is extracted as $y < 0$ (at line 4, possible reachable values of x is greater than 0 and hence possible reachable values for y is less than 0 due to the update on variable y at line 3). T_b and D_b are updated as $y > 0 \ \wedge \ y < 0$ and $y < 0$ respectively using R_b and $\mathcal{RS}(b)$ computed by PReach is 0 predicting correctly that the assertion is not reachable. Similarly, at line 6 in Fig. 3b, T_b and D_b are $x < z$ and *True* respectively. $\mathcal{S}(b)$ is computed as 0.5 predicting incorrectly that the assertion is reachable. Applying an interval analysis, there will be no refinement conditions as it is not possible to catch the relation between the variables x and z using the interval domain. But, applying relational analysis using the polyhedra domain, R_b is extracted as $x > z$ (possible reachable values of z is equal to $x - 7$). T_b and D_b are then updated as $x < z \ \wedge \ x > z$ and $x > z$ respectively and $\mathcal{RS}(b)$ is computed as 0, correctly predicting that the assertion is not reachable.

Note that, for general function invocation including recursion, it may be expensive to obtain precise interprocedural analysis, reducing the effectiveness of refinement.



(a) Control flow graph **(b) Target statement subgraph** **(c) Markov chain construction**

Figure 4: Target statement subgraph extraction and Markov chain construction for the running example

3.3 Target Statement Subgraph Extraction

The control flow graph of a program is a representation of all paths that may be traversed during execution. Given a program p, a target statement t in p and the input domain I, we extract the control flow graph of p, $\mathcal{G}(p)$, and mark the node of the control flow graph containing the target statement t as the node n^t.

We expedite our analysis by extracting the *target statement subgraph*, $\mathcal{G}(p,t)$ of $\mathcal{G}$. $\mathcal{G}(p,t)$ contains all the control flow graph information needed to perform our analysis. We define this subgraph using standard concepts from control flow analysis. We define a *branch node* b in a control flow graph to be any node with more than one outgoing edge. The corresponding *merge node* m of a branch node b is its immediate post-dominator. The *component* C defined by b is the union of branch node b, its merge node m and all nodes of the control flow graph reachable from b without going through m. The *maximal component* of a node is the largest component containing that node. Any non-maximal component containing this node will be contained in this maximal component.

To extract $\mathcal{G}(p,t)$, we first find the maximal component of n^t. If n^t is not contained in any component, then n^t must lie on every path through $\mathcal{G}(p)$. Therefore, it is reached with certainty, $\mathcal{P}(p,t) = 1$, and our analysis can be terminated. Otherwise, the maximal component of n^t is the *maximal statement subgraph*.

$\mathcal{G}(p,t)$ is a subgraph of the maximal statement subgraph. To obtain $\mathcal{G}(p,t)$, we remove any component of the maximal statement subgraph that does not contain the statement node n^t. The branch and merge nodes of these components remain in the subgraph with one outgoing edge from the branch node to the merge node. $\mathcal{G}(p,t)$ results from this procedure.

Figure 4 shows the process of the target statement subgraph extraction on the running example from Figure 2. Figure 4a gives the control flow graph $\mathcal{G}(p)$ with the statement node n^t highlighted in red. Figure 4b shows the target statement subgraph $\mathcal{G}(p,t)$ extracted from $\mathcal{G}(p)$. In this example, the branch corresponding to $y \neq 0$ is removed from the control flow graph structure. The decision made at this branch does not impact the probability of reaching the target statement node.

Note that the target statement subgraph extraction phase is a heuristic to speed up our analysis. The subsequent stages can be performed on the entire control flow graph but this would result in unnecessary work including extra model counting queries which would slow down the analysis.

3.4 Markov Chain Construction

We define a weight for each edge of $\mathcal{G}(p,t)$. These weights transform $\mathcal{G}(p,t)$ into a Discrete Time Markov Chain (DTMC), $\mathcal{M}(p,t)$. A DTMC is a tuple $(S, \bar{s}, P, L)$ where S is a finite set of states, $\bar{s} \in S$ is the initial state, $P\colon S \times S \rightarrow [0,1]$ is the transition probability matrix where $\sum_{s' \in S} P(s,s') = 1$ for all $s \in S$. Each element $P(s,s')$ of the transition probability matrix gives the probability of making a transition from state s to state s'.

We use dependency analysis in the construction of the Markov Chain as we want to identify the branches dependent on input to set the weights of the edges accordingly.

Dependency Analysis. A branch condition is input dependent if the evaluation of the condition depends on the value of the program input. Given a program and its marked input, we use static dependency analysis to identify the input dependent branches. Dependency analysis provides an over approximation of the set of branch conditions whose evaluation depends on the inputs. We use Janalyzer [5], an existing static analysis tool, to perform the dependency analysis. Janalyzer is implemented on top of the WALA [40] program analysis framework.

Then, we construct the Markov chain by assigning weights to each edge of $\mathcal{G}(p,t)$. $\mathcal{G}(p,t)$ is a directed graph: each edge begins at a source node s and ends at a destination node d. Given an edge $e : s \rightarrow d$: If e is the only edge beginning at s, the weight of e is 1. Else, s is a branch node by definition. To determine its weight we use a combination of dependency analysis and branch selectivity. Since b is a branch node, there is a branch condition associated.

- If the branch condition is independent from the program input, we weigh edge e as follows. Let E be the number of edges originating at s and $E^t \leq E$ be the number of edges originating at s which lie on a path to the target statement node n^t. If $E^t = 0$, then the weight of e is $1/E$. Otherwise, if e lies on a path to n^t weight of e is $1/E^t$. If e does not lie on a path to n^t, weight of e is 0.
- If the branch condition is dependent on the program input, we compute the weight of the edge e as follows. We use a model-counting constraint solver to determine the branch selectivity of b, $\mathcal{S}(b)$. If e is the edge corresponding to the if condition, the weight of e is $\mathcal{S}(b)$. Else, $1 - \mathcal{S}(b)$.

At the end of this phase, $\mathcal{G}(p,t)$ has been transformed into Markov chain $\mathcal{M}(p,t)$ where the probability of transitioning from one state to the next is given by the edge weight.

Figure 4c shows $\mathcal{M}(p,t)$ for the running example. The transition probabilities are given as edge weights. The two branch conditions yield the only non 1 edge weights in the graph. Both of these branch conditions are input dependent as determined by the dependency analysis. For each branch condition, the model-counting constraint solver ABC was used to find its branch selectivity. This selectivity was used to compute the weight of the edge corresponding to the if branch and its complement was used to compute the weight of the edge corresponding to the else branch.

Note that, the first-order Markov chains do not encode any context sensitivity; thus branch probabilities, e.g., loop conditions,

would always result in the same selectivity measure regardless of the call site or iteration number.

3.5 PCTL Query Formulation

We automatically synthesize queries over $\mathcal{M}(p,t)$, whose solutions yield an approximation of $\mathcal{P}(p,t)$. The query we synthesize is:

- What is the probability that the target node n^t is reached at least once?

The answer to this query approximates $\mathcal{P}(p,t)$. We use a probabilistic model checker PRISM [22], a tool that analyzes systems that exhibit probabilistic behavior, to answer this query. We generate a discrete time Markov chain (DTMC) model based on the syntax supported by the PRISM tool. We can synthesize queries like *what is the probability of reaching a state in the Markov chain eventually?*.

In PRISM, a PCTL formula is interpreted over the DTMC model. Two types of formulas are supported: state formulas and path formulas where path formulas occur only when there is a probabilistic measure that needs to be included in the specification. For our analysis, the queries we synthesize are path formulas and are of the form $P \sim p[\phi]$ which is the probabilistic analogue of the path quantifiers of CTL. For example, the PCTL formula P=?[F ϕ] states what is the probability of reaching state ϕ.

The complexity of PCTL query verification for DTMC is polynomial in the number of states [23]. Since the number of states of the DTMC is linear in the size of the program, overall complexity of PCTL query verification is polynomial of program size.

Loop Analysis. In analyzing programs which contain back edges (either from loops or from recursion), we consider two different queries for programs with loops.

- What is the probability that target node n^t is reached at least once within a given loop bound?
- What is the probability that target node n^t is reached at least once?

The first query enables us to model bounded loop executions. To answer this query, we fix a loop bound and unroll any loops in the Markov chain. If the target node n^t is duplicated during this loop unrolling process, then the query becomes

- What is the probability that any target node n^t is reached at least once?

Once the loops in the Markov Chain are unrolled, the first query becomes the initial query on the unrolled Markov chain except that there might be multiple instances of the target node.

In answering the second query, we leave the Markov chain as is including any back edges and generate the DTMC model for PRISM as it is. PRISM calculates a steady state probability for unbounded loop scenario. Bounding the loop and asking the bounded version of the reachability query under approximates the unbounded case. As the loop bound increases, the solution for the bounded case approaches that of the unbounded case and in some cases it is possible to reach the steady state probability, i.e., to reach a fixpoint. Note that, in PRISM, we are able to compute the steady state probability, so it is not necessary to compute the fixpoint by increasing loop bounds. This is one of the advantages of our approach over probabilistic symbolic execution.

4 IMPLEMENTATION

We have implemented our technique in a tool called PReach (Probabilistic Reachability Analyzer) targeting programs written in Java programming language.

Using the static analysis tool Janalyzer [5] we first extract the control flow graph from the given program. After marking inputs for which we want to calculate reachability probability, we use dependency analysis for the marked inputs and identify all input-dependent branches. We identify the target statement node and do dominator and post-dominator analysis in order to extract the target statement subgraph.

For calculating branch selectivity of input-dependent branches we first translate the branch conditions to SMT-LIB format constraints using Spoon [31] and then we use ABC [2] for model counting. To compute refined branch selectivity we applied two abstract domains, interval and polyhedra using Jana [41], a numeric analysis tool for Java. We call these implementations as PReach-I and PReach-P respectively. We define the domain size for integers as signed 31 bit, for strings as length of 16 with all printable ASCII characters, for char as unsigned 8 bit integers. Once we get the model count from ABC, we calculate the branch selectivity. To compute bounded reachability of a target statement, we look for back-edges and if there is one, we unroll the loop to a certain bound. For unbounded cases, we compute the steady state probability.

Once we have all the branch selectivity values, we construct the discrete time Markov chain (DTMC). Using the target statement node, we formulate the queries to calculate the reachability probability. We use the probabilistic model checker PRISM [22] for computing the target statement reachability probability. We convert the Markov chain to a DTMC model in PRISM syntax and synthesize queries. Then, we execute PRISM to compute the probability. We use PRISM as it provides features to reduce the reachability checking of a statement in a program with unbounded loops to reachability checking of a state in DTMC. Our current implementation determines reachability probability for each target statement separately. We can extend our approach to handle reachability of multiple statements by synthesizing slightly more complex queries.

For collecting ground truth values of *hard to reach* statements, we run a generator based random fuzzer for all the programs. We use JQF [28] tool which is a feedback directed fuzz testing platform for Java. JQF incorporates coverage-guided fuzz testing technique ZEST [29]. We use generator-based random fuzzing option provided by ZEST. We set a timeout of one hour and if the fuzzer fails to generate inputs to reach the target statement, we determine that the target statement is *hard to reach*.

Note that, the PReach approach can be extended to support alternative concrete testing techniques and the definition of *hard to reach* statements can be adapted accordingly. For example, for a random testing tool like Randoop [27] (used in the hybrid testing tool JDoop [16]), the definition of *hard to reach* can be changed by considering an input distribution that is different from uniform distribution, by using different usage profiles [18].

5 EXPERIMENTAL EVALUATION

To evaluate PReach, we experimented on benchmark programs from the Competition on Software Verification (SV-COMP) [8] and

the Competition on Software Testing (Test-Comp) [9], which we call the SV-COMP benchmark. So far, Test-Comp have only used C programs from the SV-COMP benchmark. Among the benchmarks used for Java in SV-COMP 2021, We use 4 modules (jayhorn-recursive, jbmc-regression, jpf-regression, algorithms) for evaluation. We mark all the non-deterministic inputs in the SV-COMP benchmarks as inputs for reachability analysis. We use the assert statements in these programs as target statements. We use two criteria to select the programs from these directories for our experiments. We exclude programs if one of the following two conditions hold:

(1) Target statement reachability does not depend on the inputs: PReach is not applicable for these programs as it assesses reachability probability with respect to inputs.
(2) Verification tasks are specific to floating point arithmetic: The model-counting constraint solver we use does not support constraints generated from such programs.

Based on the above criteria, our final dataset consists of a total of 142 programs. We modify these programs in order to allow us to run both our analysis and the generator based random fuzzer while keeping the program semantics unchanged. These modified programs are available at [35].

We run experiments on a virtual box equipped with an Intel Core i7-8750H CPU at 2.20GHz and 16 GB of RAM running Ubuntu Linux 18.04.3 LTS and the Java 8 Platform Standard Edition, version 1.8.0_232, from OpenJDK 64-Bit Server VM.

5.1 Results for the SV-COMP benchmark

Reachability probability computed by PReach is a value between 0 and 1. In order to assess how good PReach is to identify *hard to reach* statements, we classify program statements to two groups: *hard to reach* and *easy to reach*. As ground truth, we classify the programs for which the random fuzzer is unable to reach the target statement within the given time bound as *hard to reach*. We list the number of true positives (TP: ground truth is *hard to reach* and PReach predicts *hard to reach*); false positives (FP: ground truth is *easy to reach* and PReach predicts *hard to reach*); true negatives (TN: ground truth is *easy to reach* and PReach predicts *easy to reach*); false negatives (FN: ground truth is *hard to reach* and PReach predicts *easy to reach*). A *hard to reach* threshold (T_H) value 0.05 means statements having reachability probability less than 0.05 are classified as *hard to reach*. Then, we evaluate PReach with respect to the ground truth.

Table 1 shows the overall precision, recall and accuracy results of PReach-P. Precision, recall and accuracy for different implementations of PReach is shown in Table 4. We demonstrate results for multiple values of T_H to analyze changes in precision, recall and accuracy across the benchmarks. Reducing T_H from 0.05 to 0.01 does not change the results at all. Increasing T_H to 0.1 leads to interesting changes in the results: some of the true negative cases are updated to false positives, reducing precision and accuracy. Increasing T_H to 0.25 changes the results further: the number of false positive cases are increased and number of true negative cases are decreased. Increasing the value of T_H changes the prediction of more cases from *easy to reach* to *hard to reach* and hence, the overall precision is reduced from 95.8% to 79.3% and the overall accuracy is reduced from 95.1% to 88.0%. The ability of using different threshold values demonstrates the quantitative nature of our analysis rather than being a fixed binary classification.

Accuracy of PReach-P setting T_H as 0.05 or 0.01 is 95.1%. Across all the benchmarks, accuracy is greater than or equal to 87.0%, reflecting the effectiveness of our heuristic. PReach-P fails to identify 5 of the *hard to reach* program statements having a recall of 90.2%, but it is very precise in identifying *hard to reach* program statements with a precision of 95.8%.

Among 142 cases, only 2 cases are false positives and 5 cases are false negatives. The remaining 135 cases are correctly classified by PReach. The reasons behind the 2 false positive cases and the 5 false negative cases are: 1) most of the input values generated by the fuzzer lead to exceptions and the fuzzer cannot generate enough valid inputs, 2) the numeric analysis tool cannot handle complex operations such as multiplication, division and modulus between more than one variables using the abstract domains.

Experimental results show that among the 3 variations of the tools, PReach-P performs the best with a precision, recall and accuracy of 95.8%, 90.2% and 95.1% respectively. Without applying refined branch selectivity, PReach cannot catch two scenarios: 1) two dependent branch conditions cancel out each other, 2) input values are updated in a way that the branch condition becomes always true or false. Hence, the number of false negatives increases from 5 to 13. PReach-I uses interval domain for refinement analysis which is not as precise as PReach-P using a plyhedra domain. As a result, 2 extra false negatives are introduced by PReach-I.

5.2 Probabilistic Symbolic Execution (PSE)

We provide an experimental comparison of PReach with probabilistic symbolic execution (PSE) [20]. We use SPF [30] as the symbolic execution engine for PSE. PSE is unable to analyze some of the target programs due to unsupported constraints such as non-linear path constraints, PReach does not face this issue as much since it only considers branch conditions. The rest of the programs are marked as analyzable by PSE, as shown in Table 2. For programs where the number of recursive calls or loop iterations depend on the input, PSE can not explore all possible paths since it can only search programs behaviors up to a bounded execution depth (search depth), and since the number of program paths grows exponentially. Therefore, we set a timeout of 1 hour for PSE and evaluate for different search depths. Since PSE is unable to cover all program paths, the probabilistic measurement computed by PSE is not exact. Increasing the search depth allows PSE to obtain more accurate results but also increases the number of program paths exponentially. This leads PSE to time out for some programs, as shown in Table 2. This is not the case for the jpf-regression and jbmc-regression benchmarks, as there is no input dependent recursive calls or loops.

We show the comparison of reachability probabilities computed by PReach and PSE in Table 3. As we do not have any ground truth for the probability measurement, we calculate probability differences between these two techniques and analyze the differences in case of agreement and disagreement for *hard to reach* statement assessment. PReach and PSE agree if their predictions match, disagree otherwise. Based on agreement and disagreement, we divide all the cases into 3 groups: 1) agreement, 2) disagreement

Table 1: Effectiveness of PReach-P in terms of precision, recall and accuracy scores for sv-comp benchmarks

Benchmarks	Threshold (T_H)																			
	0.25							0.1							0.05/0.01					
	TP	FP	TN	FN	Precision	Recall	Accuray	TP	FP	TN	FN	Precision	Recall	Accuracy	TP	FP	TN	FN	Precision	Recall	Accuracy
jayhorn-recursive	9	1	10	3	90.0	75.0	82.6	9	0	11	3	100.0	75.0	87.0	9	0	11	3	100.0	75.0	87.0
jpf-regression	25	7	43	2	78.1	92.6	88.3	25	2	48	2	92.6	92.6	94.8	25	2	48	2	92.6	92.6	94.8
jbmc-regression	8	1	12	0	88.8	100.0	100.0	8	0	13	0	100.0	100.0	100.0	8	0	13	0	100.0	100.0	100.0
algorithms	4	3	14	0	57.1	100.0	85.7	4	3	14	0	57.1	100.0	85.7	4	0	17	0	100.0	100.0	100.0
Total	46	12	79	5	79.3	90.2	88.0	46	5	86	5	90.2	90.2	93.0	46	2	89	5	95.8	90.2	95.1

Table 2: Number of programs analyzed by PReach and Probabilistic Symbolic Execution within 1 hour timeout

Benchmarks	Number of programs analyzed								
	PReach	Probabilistic Symbolic Execution							
		Analy-zable	Analyzable with Search Depth						
			10	20	30	100	500	1000	∞
jayhorn-recursive	23	21	21	17	11	6	5	1	1
jpf-regression	77	69	69	69	69	69	69	69	69
jbmc-regression	21	16	16	16	16	16	16	16	16
algorithms	21	9	9	9	9	9	8	6	2
Total	142	115	115	111	105	100	98	92	88

Table 3: Probabilistic measurement differences and *hard to reach* statement prediction disagreements between PReach (PR) and PSE

Bench-marks	Search Depth	#Cases Analy-zable	Tool	Agreement		Disgreement				All Cases
						PSE ✓		PReach ✓		
				#	Avg. Diff.	#	Avg. Diff.	#	Avg. Diff.	Average Diff.
jayhorn-recursive	10	21	PR	16	0.086	2	1.000	3	0.420	0.270
			PR-I	16	0.086	2	1.000	3	0.420	0.270
			PR-P	16	0.086	2	1.000	3	0.420	0.270
jpf-regression	∞	69	PR	58	0.050	10	0.550	1	0.250	0.083
			PR-I	62	0.049	6	0.542	1	0.250	0.095
			PR-P	64	0.035	4	0.625	1	0.250	0.072
jbmc-regression	∞	16	PR	14	0.040	2	0.250	0	-	0.066
			PR-I	16	0.031	0	-	0	-	0.031
			PR-P	16	0.031	0	-	0	-	0.031
algorithms	100	9	PR	3	0.087	0	-	6	0.390	0.317
			PR-I	3	0.087	0	-	6	0.390	0.317
			PR-P	3	0.087	0	-	6	0.390	0.317

and PSE is correct, 3) disagreement and PReach is correct. The average difference in probability is low for the cases of agreement. The difference is even lower for jpf-regression and jbmc-regression benchmarks as PSE achieves very high precision and accuracy (see Table 5) and PReach agrees with the predictions. For the cases of disagreement, the difference is very high for most of the cases when PSE predicts correctly but PReach does not. One of the main reasons for this is variable updates making some of the program paths infeasible. PSE can catch the infeasible paths whereas PReach gives an approximate result for these cases using branch selectivity. Both PReach-I and PReach-P can address this issue. Using refined branch selectivity, the number of agreement cases are increased and average probability difference is reduced for jpf-regression and jbmc-regression benchmarks. Another reason is PReach predicting a program statement as *easy to reach* but the ground truth is *hard to reach* as fuzzer cannot reach the target statement due to recursion stack overflow error. Average difference is also high for jayhorn-recursive and algorithms benchmarks when PReach predicts correctly but PSE does not, as there is an exponential increase in the number of paths and PSE poorly approximates the probability.

We now compare these two techniques in terms of *hard to reach* statement prediction accuracy and precision. To compare PReach and PSE, we set the *hard to reach* threshold to 0.05. Table 4 shows precision, recall and accuracy for PReach and PSE with search depth 10 and 1000. We evaluate all 142 programs analyzable by PReach. The programs for which PSE times out are marked as *easy to reach* as our target is to identify the *hard to reach* program statements. Different search depths do not change results for jpf-regression and jbmc-regression benchmarks as these programs are free of recursive calls and loops that depend on inputs. The precision and accuracy values for PReach are comparable to PSE for these benchmarks. The prediction results are improved a lot using PReach-I and PReach-P. For jpf-regression and jbmc-regression benchmarks, precision, recall and accuracy are increased. For jbmc-regression benchmarks, both PReach-I and PReach-P performs better than PSE and for jpf-regression benchmarks, overall scores achieved by PReach-P are better than PReach-I and very close to the scores achieved by PSE. For jayhorn-recursive and algorithms benchmarks, PSE can not achieve as good results as PReach, PReach-I or PReach-P since these programs need to deal with input dependent recursive calls and loops. For lower search depth (10), PSE can not explore all the program paths and as a result the computed probability is an under-approximation (worse than a heuristics-based approach used in PReach). For higher search depth (1000), most of the programs time out and hence are marked as *easy to reach*. As a result there are no true-positive cases making precision and recall values 0 as well as no false-positive cases keeping the total precision high (96.9). For the algorithms benchmark, even with search depth 10, the precision and recall is 0 as PSE can not support most of the programs (marked as *easy to reach*) as array size is input dependent and marked as symbolic, which is not analyzable by SPF. Though for programs with bounded execution depth due to the absence of loop and recursion (jpf-regression and jbmc-regression benchmarks), PSE performs better than PReach but PReach-P is as good as or even better in some cases than PSE.

We show precision and accuracy for the 85 programs in these two benchmarks that are analyzable by PSE in Table 5. The scores for PSE are not 100% due to situations like integer arithmetic overflow that are not caught by symbolic execution. The precision (95.7) and accuracy (87.1) for PReach is comparable to PSE and is impressive given that it is a scalable heuristic approach. The precision (96.8) and accuracy (96.5) by PReach-P is very close to the scores achieved by PSE. Moreover, PSE performs very poorly on programs with unbounded execution depth (jayhorn-recursive and algorithms

benchmarks) whereas PReach, PReach-I and PReach-P have high precision and accuracy.

Table 6 shows the average analysis time required and percentage of cases analyzed by both of these techniques. Even for a low search depth (10) the analysis time of PSE is higher than PReach. Note that, lower search depths in PSE poorly approximate the probability. However, increasing the search depth increases the analysis time by orders of magnitude. For both jayhorn-recursive and algorithms benchmarks, the average analysis time increases and percentage of analyzed cases within the time bound decreases as the search depth is increased. For the jayhorn-recursive benchmark, even for a search depth of 30 the average analysis time increases by an order of magnitude. This is because the number of recursive function calls are input dependent. The average analysis time shown in the table is less than or equal to 3600 seconds since we set the timeout to 1 hour (i.e., 3600 seconds is the maximum analysis time). The time for jayhorn-recursive benchmarks with search depth greater than or equal to 30 would be very high without this timeout. Average analysis time also increases for the algorithms benchmarks when the search depth is increased as number of loop iterations depend on the inputs. These results show that PSE is not scalable for unbounded execution depth whereas PReach is.

PReach-I and PReach-P require more analysis time compared to PSE for jpf-regression and jbmc-regression benchmarks. As programs in these benchmarks are loop and recursion free, PSE runs fast whereas PReach-I and PReach-P perform abstract interpretation for branch selectivity refinement. However, as the search depth of the programs increases, the branch selectivity refinement analysis time becomes less significant compared to the exponential time increase due to path constraint solving performed by PSE, reflected in the jayhorn-recursive and algorithms benchmarks. For these benchmarks, as the search depth increases to 100, the analysis time by PSE is orders of magnitude higher than the analysis time required by PReach-I or PReach-P. These results clearly indicate that PReach, PReach-I and PReach-P maintain a balanced trade off between precision and scalability for probabilistic reachability analysis and among these three implementations, PReach-P performs the best considering its high precision and accuracy.

5.3 Statistical Symbolic Execution (SSE)

In this section, we provide an experimental comparison of PReach-P with statistical symbolic execution (SSE). Prior work has demonstrated that SSE is more precise and faster than PSE when large execution bounds are necessary, preventing PSE from terminating [18]. SSE uses SPF [30] as the symbolic execution engine similar to PSE. We compare PReach-P and SSE only for the jayhorn-recursive and algorithms benchmarks from SV-COMP, as PSE achieves very high precision and accuracy for jpf-regression and jbmc-regression benchmarks, and we have already compared the performance of PReach-P and PSE on those benchmarks.

SSE is unable to analyze 12 out of 44 target programs due to inability to handle non-linear path constraints or symbolic array indexing during symbolic execution. As before, we set a timeout of 1 hour for SSE and evaluate for different search depths. Like PSE, SSE is also unable to explore all program paths within an hour, but it can provide statistical guarantees for the computed probabilities with respect to accuracy (ϵ) and confidence (δ) parameters [18]. SSE has two different sampling approaches: 1) Monte Carlo and 2) Informed sampling. We compare PReach-P to both of these sampling techniques in SSE. In both cases, we set ϵ to be 10^{-5} and target δ to be 0.99 following the experimental setup in [18]. For Monte Carlo sampling, we set the maximum sample size (N_1) as 100, 000 and for informed sampling, we set N_1 as 100 and maximum number of iterations as 100.

Precision, recall and accuracy for SSE is presented in Table 7. SSE has better precision, recall and accuracy compared to PSE but not compared to PReach-P. Recall and accuracy for SSE drops with increasing search depth. For algorithms, precision and recall is 0.0 (marked with a *), as there were no true positive cases among the analyzable programs by SSE. Similar to the experimental setup for the comparison to PSE, we mark a program statement as *easy to reach* if it times out.

We do not take the reported statistical confidence into account to determine which program statements should be marked as *hard to reach* or *easy to reach* by SSE. One could use a threshold value for the statistical confidence, and accept only the predictions achieving a certain confidence. In that case, the precision and accuracy of SSE would drop further. Instead, we present average confidence achieved by SSE in Table 8 separately. Statistical confidence achieved by SSE drops as the search depth for symbolic execution is increased and more programs time out. Even though we set a large maximum number of samples (100,000) for Monte Carlo sampling, SSE can not achieve a high confidence. On the other hand, informed sampling can achieve high confidence with search depths 10 or 100 for some cases. But, with an infinite search depth, none of the sampling techniques can achieve high confidence.

Average analysis time for SSE is presented in Table 8. In general, PReach-P is orders of magnitude faster than SSE. Monte Carlo sampling is consistently slower for all the programs compared to PReach-P. Informed sampling performs much better than Monte Carlo sampling. Analysis time of SSE with informed sampling is close to PReach-P for some programs when a short search depth value is used. But, irrespective of search depth, for a good number of programs, informed sampling is also orders of magnitude slower than PReach-P, and hence its average analysis time is significantly higher than PReach.

These results demonstrate that PReach-P is more scalable compared to SSE and achieves better precision and accuracy, especially for programs containing large number of paths.

5.4 Case Studies

In this section, we evaluate the effectiveness of PReach to detect *hard to reach* program statements in larger projects. We are particularly interested in program points where inputs need to pass through numerous branches to reach. We selected a set of methods from Apache Commons Lang [1] and DARPA STAC Benchmarks [4] and identified target program statements. We have analyzed 24 program statements in 12 methods from Apache Commons Lang project and 12 program statements from 6 methods across 5 projects from DARPA STAC Benchmarks.

Table 9 shows PReach results for the selected 24 cases. First, we run PSE to compute reachability probability on all these cases.

Table 4: Precision, Recall and Accuracy of PReach (PR) and PSE, computed for 142 programs, program is marked *easy to reach* if analysis times out

Bench-marks	Precision					Recall					Accuracy				
	PR	PR-I	PR-P	PSE with Search Depth		PR	PR-I	PR-P	PSE with Search Depth		PR	PR-I	PR-P	PSE with Search Depth	
				10	1000				10	1000				10	1000
jayhorn-recursive	100.0	100.0	100.0	76.9	*0.0	75.0	75.0	75.0	83.3	*0.0	87.0	87.0	87.0	78.3	47.8
jpf-regression	90.5	92.0	92.6	96.2	96.2	70.4	85.2	92.6	96.2	96.2	87.0	92.2	94.8	97.4	97.4
jbmc-regression	100.0	100.0	100.0	100.0	100.0	75.0	100.0	100.0	75.0	75.0	90.5	100.0	100.0	90.5	90.5
algorithms	100.0	100.0	100.0	*0.0	*0.0	100.0	100.0	100.0	*0.0	*0.0	100.0	100.0	100.0	52.4	61.9
Total	95.0	95.7	95.8	80.4	96.9	74.5	86.3	90.2	82.0	62.0	89.4	93.7	95.1	79.6	85.9

Table 5: Precision, Recall and Accuracy of PSE and PReach (PR), out of 85 programs computed within 1 hour for jpf- and jbmc-regression benchmarks

Bench-marks	Precision				Recall				Accuracy			
	PR	PR-I	PR-P	PSE	PR	PR-I	PR-P	PSE	PR	PR-I	PR-P	PSE
jpf-regression	94.7	95.7	96.0	96.2	69.2	84.6	92.3	100.0	87.0	92.8	95.7	98.6
jbmc-regression	100.0	100.0	100.0	100.0	66.7	100.0	100.0	100.0	87.5	100.0	100.0	100.0
Total	95.7	96.6	96.8	96.9	68.8	87.5	93.8	100.0	87.1	94.1	96.5	98.8

Table 6: Average Analysis Time for PReach (PR) and PSE, maximum average analysis time is limited to 3600 seconds, cases with timeout are included

Bench-marks	Average Analysis time in seconds (% Cases Analyzed in 1 hour)						
	PR	PR-I	PR-P	Probabilistic Symbolic Execution Search Depth			
				10	30	100	1000
jayhorn-recusrive	2.43	4.35	6.16	5.34 (91%)	2048.33 (52%)	2583.22 (29%)	3428.67 (5%)
jpf-regression	0.81	3.11	4.86	1.51 (91%)	1.51 (91%)	1.5 (91%)	1.51 (91%)
jbmc-regression	0.69	4.90	6.10	3.32 (76%)	3.32 (76%)	3.32 (76%)	3.32 (76%)
algorithms	0.99	6.38	9.69	2.25 (43%)	3.98 (43%)	79.76 (43%)	2399.94 (29%)
Total	1.08	4.08	5.97	2.51 (82%)	372.50 (75%)	475.21 (71%)	808.28 (65%)

Among 18 methods we analyze we find that PSE is not able to handle 9 methods due to either variable type conversion or lack of support for some String library functions. PSE fails on 2 other methods due to incapability to model count for non-linear path constraints and another 4 methods due to lack of support for translation of expressions to string path constraints. PReach does not have any of these issues as the underlying technique is simpler than symbolically executing a program, and it can avoid dealing with non-linear path constraints and complex string path constraints as it needs to consider individual branch conditions only. Finally, PSE successfully runs on 3 methods but for 2 of the methods it times out, predicting only 1 case correctly as *hard to reach*. These results demonstrate the limitations and poor scalability of probabilistic symbolic execution on realistic programs. We also cannot analyze these cases using PReach-I and PReach-P as the programs perform string operations and the abstract interpretation tool [41] we use for computing refined branch selectivity is limited to numeric analysis. Even without refining the branch selectivity, our results for these case studies demonstrate that even the base technique (PReach) using branch selectivity is capable of predicting *hard to reach* program statements efficiently for sizable programs.

PReach can predict 19 out of 24 cases correctly with an accuracy of 79.2% setting T_H as 0.001. We used the same value of T_H across all domains. Different values of T_H for Integer/mixed domain (0.01) and String domain (0.001) increases the accuracy to 83.33% supporting the quantitative nature of our analysis. 5 of the cases that PReach can not predict correctly is due to the similar reasons as SV-COMP benchmarks. The value of the input is updated inside the program and as a result the following branches do not depend on the initial input value anymore.

6 RELATED WORK

There has been an increasing amount of research on quantitative program analysis techniques based on model counting constraint solvers, and there has been a surge of progress in model counting constraint solvers [2, 11, 13, 14, 24, 25]. Model counting constraint solvers have been used in a variety of quantitative program analysis tasks such as probabilistic analysis [10, 18, 20], reliability analysis [17], estimating performance distribution [15], quantitative information flow [3, 6, 19, 33, 34], and side-channel attack synthesis [7, 32, 36]. Branch selectivity and probabilistic reachability heuristic we introduce in this paper are fundamental quantitative program analysis techniques and rely on the recent developments in model counting constraint solvers.

Probabilistic symbolic execution [20] and statistical symbolic execution [18] can be used for probabilistic reachability analysis problem we study in this paper. However probabilistic symbolic execution suffers from path explosion [12] and increasing size of path constraints with increasing execution depth, which can lead to double exponential blow up. Moreover, probabilistic symbolic execution can only analyze program behaviors up to a fixed execution depth. Statistical symbolic execution is more efficient compared to probabilistic symbolic execution but still suffers with increasing execution depth. The approach we present in this paper using branch selectivity addresses these issues since it does not suffer from path explosion and it analyzes branch conditions instead of path constraints modeling behaviors of arbitrarily long paths.

Hybrid testing techniques [16, 26, 38, 39, 42] combine concrete and symbolic techniques in order to improve effectiveness of testing. Strategy function for hybrid testing need to decide when to apply concrete techniques and when to apply symbolic. Existing techniques assess the difficulty of concrete testing to do make the decision based on the saturation of random testing [26, 38] or using a predefined configuration of time to run for concrete and symbolic techniques [16] or probabilistic program analysis [39, 42]. Markov decision process construction extracting control flow graph and

Table 7: Precision, Recall and Accuracy of PReach-P (PR-P) and SSE, computed for 44 programs from jayhorn-recursive and algorithms benchmarks, program is marked *easy to reach* if analysis times out(1 hour), both Monte Carlo and informed sampling has same precision, recall and accuracy

Benchmarks	Precision				Recall				Accuracy			
	PR-P	SSE with Search Depth			PR-P	SSE with Search Depth			PR-P	SSE with Search Depth		
		10	100	∞		10	100	∞		10	100	∞
jayhorn-recursive	100.0	100.0	100.0	100.0	75.0	75.0	75.0	58.3	87.0	87.0	87.0	78.3
algorithms	100.0	0.0*	0.0*	0.0*	100.0	0.0*	0.0*	0.0*	100.0	71.4	71.4	71.4
Total	100.0	100.0	100.0	100.0	83.3	50.0	50.0	38.9	93.2	79.5	79.5	75.0

Table 8: Average Analysis Time and statistical Confidence (δ) for PReach-P (PR-P) and SSE Monte Carlo (MCS) and informed (IS) sampling, maximum average analysis time is limited to 3600 seconds, cases with timeout are included, confidence is set to 0.0 for timeout cases

Bench-marks	Average Analysis Time							Statistical Confidence (δ)					
	PR-P	SSE-MCS with Search Depth			SSE-IS with Search Depth			SSE-MCS with Search Depth			SSE-IS with Search Depth		
		10	100	∞	10	100	∞	10	100	∞	10	100	∞
jayhorn-recursive	6.16	1495.60	2117.46	2530.45	165.71	362.36	2038.48	0.061	0.039	0.032	0.957	0.913	0.435
algorithms	9.69	2558.30	3066.67	3071.77	2004.41	2008.00	2802.04	0.016	0.016	0.016	0.444	0.444	0.222
Total	7.84	2002.91	2577.38	2786.37	1046.43	1150.83	2406.93	0.040	0.028	0.025	0.712	0.690	0.333

Table 9: Case study of PReach on Apache Commons Lang and DARPA STAC Benchmarks. PReach predicts a statement as *hard to reach* if reachability probability is less than 0.001. 19 out of 24 cases are predicted correctly

Project	Class Name	Method Name	Target Statement Line Number	Number of Branches in Method	Max #Branches to Target Statement	Reachability Probability	Random Fuzzer Ground Truth	PReach Prediction	Prediction Match
apache-commons-lang	Fraction	greatestCommonDivisor	595	11	7	0.00%	Yes	Yes	✓
	NumberUtils	createNumber	759	31	25	0.00%	No	Yes	✗
		isCreatable	1690	25	23	0.33%	No	No	✓
	FastDatePrinter	parseToken	363	7	7	7.32%	No	No	✓
	StrTokenizer	readWithQuotes	804	8	8	0.00%	Yes	Yes	✓
	StrSubstitutor	substitute	837	17	13	0.00%	Yes	Yes	✓
	NumericEntityUnescaper	translate	107	9	4	0.08%	No	Yes	✗
	ArrayUtils	shift	6994	9	9	0.00%	No	Yes	✗
	BooleanUtils	toBooleanObject	650	15	15	0.00%	Yes	Yes	✓
	RandomStringUtils	random	427	16	16	0.00%	Yes	Yes	✓
	StringUtils	containsAny	1248	8	7	0.00%	Yes	Yes	✓
	CharSequenceUtils	regionMatches	377	7	7	0.00%	Yes	Yes	✓
calculator_3	RomanNumeralFormatter	parseObject	130	34	17	0.15%	No	No	✓
			152		11	49.44%	No	No	✓
			170		34	0.59%	Yes	No	✗
calculator_4	Arithmetizer	assessParentheses	213	9	4	0.19%	No	No	✓
			245		9	93.75%	No	No	✓
	ImperialFormatter	parseObject	70	11	4	1.37%	No	No	✓
			96		9	24.63%	No	No	✓
			103		11	13.99%	No	No	✓
emu6502	Assembler	assembleLine	214	22	9	0.00%	No	Yes	✗
			207		8	0.15%	No	No	✓
linear_algebra_platform	MatrixSerializer	readMatrixFromCSV	51	13	9	7.30%	No	No	✓
rsa_commander	DecInputStream	read	99	19	12	0.08%	Yes	Yes	✓

putting probabilities as edge weight has been used to find optimal strategy for concolic testing [39]. Probabilistic path prioritization is used in [42] to decide when to invoke symbolic execution in hybrid fuzzing. Our approach focuses on identifying *hard to reach* statements based on probabilistic reachability heuristic.

7 CONCLUSIONS

We presented a novel heuristic for probabilistic reachability analysis to identify *hard to reach* program statements that uses dependency analysis, model counting, abstract interpretation, and probabilistic model checking to compute probability of reaching a program statement given random inputs. We experimentally evaluated our approach on a set of benchmark programs and demonstrated that our approach can identify statements that are *hard to reach* with reasonable precision and accuracy. We provided detailed comparison of our approach against probabilistic symbolic execution and statistical symbolic execution, demonstrating that our approach is more efficient and scalable.

ACKNOWLEDGEMENT

We would like to thank all the reviewers for their useful technical comments and insightful suggestions towards improving this paper.

REFERENCES

[1] Apache Commons Lang. 2020. https://commons.apache.org/proper/commons-lang/.

[2] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-Based Model Counting for String Constraints. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)*. 255–272.

[3] Michael Backes, Boris Köpf, and Andrey Rybalchenko. 2009. Automatic Discovery and Quantification of Information Leaks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*. 141–153.

[4] Daniel Balasubramanian, Kasper Luckow, Corina Păsăreanu, Abdulbaki Aydin, Lucas Bang, Tevfik Bultan, Miroslav Gavrilov, Temesghen Kahsai, Rody Kersten, Dmitriy Kostyuchenko, Quoc-Sang Phan, Zhenkai Zhang, and Gabor Karsai. 2017. ISSTAC: Integrated Symbolic Execution for Space-Time Analysis of Code.

[5] Daniel Balasubramanian, Zhenkai Zhang, Dan McDermet, and Gabor Karsai. 2017. Janalyzer: A Static Analysis Tool for Java Bytecode. *ISIS* 17 (2017), 104.

[6] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Păsăreanu, and Tevfik Bultan. 2016. String Analysis for Side Channels with Segmented Oracles. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA.

[7] Lucas Bang, Nicolas Rosner, and Tevfik Bultan. 2018. Online Synthesis of Adaptive Side-Channel Attacks Based On Noisy Observations. In *Proceedings of the IEEE European Symposium on Security and Privacy*.

[8] Dirk Beyer. 2021. Software verification: 10th comparative evaluation (SV-COMP 2021). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 401–422.

[9] Dirk Beyer. 2021. Status report on software testing: Test-Comp 2021. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, Cham, 341–357.

[10] Mateus Borges, Antonio Filieri, Marcelo d'Amorim, and Corina S. Păsăreanu. 2015. Iterative distribution-aware sampling for probabilistic symbolic execution. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*. 866–877.

[11] Mateus Borges, Quoc-Sang Phan, Antonio Filieri, and Corina S. Păsăreanu. 2017. Model-Counting Approaches for Nonlinear Numerical Constraints. In *Proceedings of the 9th International NASA Formal Methods Symposium*. 131–138.

[12] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. *Commun. ACM* 56, 2 (2013), 82–90.

[13] Supratik Chakraborty, Daniel J. Fremont, Kuldeep S. Meel, Sanjit A. Seshia, and Moshe Y. Vardi. 2014. Distribution-Aware Sampling and Weighted Model Counting for SAT. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*. 1722–1730.

[14] Supratik Chakraborty, Kuldeep S. Meel, Rakesh Mistry, and Moshe Y. Vardi. 2016. Approximate Probabilistic Inference via Word-Level Counting. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*. 3218–3224.

[15] Bihuan Chen, Yang Liu, and Wei Le. 2016. Generating performance distributions via probabilistic symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 49–60.

[16] Marko Dimjašević, Falk Howar, Kasper Luckow, and Zvonimir Rakamarić. 2018. Study of integrating random and symbolic testing for object-oriented software. In *International Conference on Integrated Formal Methods*. Springer, 89–109.

[17] Antonio Filieri, Corina S. Pasareanu, and Willem Visser. 2013. Reliability analysis in symbolic pathfinder. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. 622–631.

[18] Antonio Filieri, Corina S. Păsăreanu, Willem Visser, and Jaco Geldenhuys. 2014. Statistical Symbolic Execution with Informed Sampling. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 437–448. https://doi.org/10.1145/2635868.2635899

[19] Daniel J. Fremont and Sanjit A. Seshia. 2014. Speeding Up SMT-Based Quantitative Program Analysis. In *In 12th International Workshop on Satisfiability Modulo Theories (SMT)*.

[20] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*. 166–176.

[21] Bertrand Jeannet and Antoine Miné. 2009. Apron: A library of numerical abstract domains for static analysis. In *International Conference on Computer Aided Verification*. Springer, 661–667.

[22] Marta Kwiatkowska, Gethin Norman, and David Parker. 2004. Probabilistic symbolic model checking with PRISM: A hybrid approach. *International Journal on Software Tools for Technology Transfer* 6, 2 (2004), 128–142.

[23] Marta Kwiatkowska, Gethin Norman, and David Parker. 2007. Stochastic model checking. In *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 220–270.

[24] Jesús A. De Loera, Raymond Hemmecke, Jeremiah Tauzer, and Ruriko Yoshida. 2004. Effective lattice point counting in rational convex polytopes. *Journal of Symbolic Computation* 38, 4 (2004), 1273 – 1302. https://doi.org/10.1016/j.jsc.2003.04.003

[25] Loi Luu, Shweta Shinde, Prateek Saxena, and Brian Demsky. 2014. A model counter for constraints over unbounded strings. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 57.

[26] Rupak Majumdar and Koushik Sen. 2007. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 416–426.

[27] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *OOPSLA 2007 Companion, Montreal, Canada*. ACM.

[28] Rohan Padhye, Caroline Lemieux, and Koushik Sen. 2019. JQF: Coverage-Guided Property-Based Testing in Java. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'19). https://doi.org/10.1145/3293882.3339002*.

[29] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic Fuzzing with Zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 329–340. https://doi.org/10.1145/3293882.3330576

[30] Corina S Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 179–180.

[31] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. 2016. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience* 46, 9 (2016), 1155–1179.

[32] Quoc-Sang Phan, Lucas Bang, Corina S. Pasareanu, Pasquale Malacaria, and Tevfik Bultan. 2017. Synthesis of Adaptive Side-Channel Attacks. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. 328–342.

[33] Quoc-Sang Phan, Pasquale Malacaria, Corina S. Păsăreanu, and Marcelo d'Amorim. 2014. Quantifying information leaks using reliability analysis. In *Proceedings of the International Symposium on Model Checking of Software, SPIN 2014, San Jose, CA, USA*. 105–108.

[34] Quoc-Sang Phan, Pasquale Malacaria, Oksana Tkachuk, and Corina S. Păsăreanu. 2012. Symbolic quantitative information flow. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[35] Seemanta Saha. 2022. PReach: A probabilistic reachability analyzer to identify hard to reach program statements. https://zenodo.org/record/5915206.

[36] Seemanta Saha, William Eiers, Ismet Burak Kadron, Lucas Bang, and Tevfik Bultan. 2019. Incremental Attack Synthesis. *ACM SIGSOFT Software Engineering Notes* 44, 4 (2019), 16–16.

[37] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2017. Fast polyhedra abstract domain. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 46–59.

[38] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*.

[39] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards optimal concolic testing. In *Proceedings of the 40th International Conference on Software Engineering*. 291–302.

[40] IBM Watson. 2006. Watson libraries for analysis. wala. sourceforge. net/wiki/index. php. *Main Page* (2006).

[41] Shiyi Wei, Piotr Mardziel, Andrew Ruef, Jeffrey S Foster, and Michael Hicks. 2018. Evaluating design tradeoffs in numeric static analysis for java. In *European Symposium on Programming*. Springer, Cham, 653–682.

[42] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing.. In *NDSS*.