Statistical Reachability Analysis

Seongmin Lee
Max Planck Institute for Security and Privacy
Bochum, Germany
seongmin.lee@mpi-sp.org

Marcel Böhme
Max Planck Institute for Security and Privacy
Bochum, Germany
marcel.boehme@acm.org

ABSTRACT

Given a target program state (or statement) s, what is the probability that an input reaches s? This is the quantitative reachability analysis problem. For instance, quantitative reachability analysis can be used to approximate the reliability of a program (where s is a bad state). Traditionally, quantitative reachability analysis is solved as a model counting problem for a formal constraint that represents the (approximate) reachability of s along paths in the program, i.e., probabilistic reachability analysis. However, in preliminary experiments, we failed to run state-of-the-art probabilistic reachability analysis on reasonably large programs.

In this paper, we explore statistical methods to estimate reachability probability. An advantage of statistical reasoning is that the size and composition of the program are insubstantial as long as the program can be executed. We are particularly interested in the error compared to the state-of-the-art probabilistic reachability analysis. We realize that existing estimators do not exploit the inherent structure of the program and develop structure-aware estimators to further reduce the estimation error given the same number of samples. Our empirical evaluation on previous and new benchmark programs shows that (i) our statistical reachability analysis outperforms state-of-the-art probabilistic reachability analysis tools in terms of accuracy, efficiency, and scalability, and (ii) our structure-aware estimators further outperform (blackbox) estimators that do not exploit the inherent program structure. We also identify multiple program properties that limit the applicability of the existing probabilistic analysis techniques.

CCS CONCEPTS

• **Theory of computation** → **Program analysis**; • **Mathematics of computing** → *Bayesian computation.*

KEYWORDS

Quantitative reachability analysis, Statistical reachability analysis, Reaching probability, Markov chain

ACM Reference Format:
Seongmin Lee and Marcel Böhme. 2023. Statistical Reachability Analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3611643.3616268

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616268

1 INTRODUCTION

The traditional assessment of the reachability of a program state provides only a true-false answer: either the state is reachable (e.g., the program may crash for some input) or not (e.g., it never crashes for any input). Due to the undecidability of the analysis problem [16] and the restricted expressiveness of the analysis result, such a binary answer provides only limited information. Instead of a binary answer, *quantitative reachability analysis* provides the probability of how likely a certain program state is reached given the workload of the program. Such a *quantitative* measure of reachability can provide more comprehensive information about the program semantics. For instance, it can estimate how probable is to reach a crashing state under normal workload, which can be critical information for software reliability/security/maintenance.

The typical method considered for quantitative reachability analysis is called *probabilistic reachability analysis* [26], which *analytically computes* the reaching probability directly from the source code. Probabilistic Symbolic Execution (PSE), the pioneering work by Geldenhuys et al. [12], computes the reaching probability of a program state by finding all the path conditions to reach the state using symbolic execution and counting the number of inputs satisfying the path conditions using model counting; the sum of the probabilities becomes the exact reaching probability of the program state. As PSE may suffer from scalability issues for a large and complex program, many follow-up works have been proposed to improve the scalability of probabilistic reachability analysis [11, 13]. Most recently, Saha et al. proposed PReach which computes the reaching probability using branch-level probability information [26].

When facing a problem too complex for the analytical method, especially when it is unmanageable to compute a quantity exactly, a sampling-based statistical method can be used to overcome the limitation [4]. It is well-known that Monte Carlo methods have been successfully applied to numerous problems across various fields, including natural sciences [10] and engineering [22], where the solution is intractable for analytic computation. Recently, in the context of program analysis, Liyanage et al. [20] proposed a statistical method to approximate the number of elements that can be reached by actual program execution, which, previously, can only be upper-bounded by static analysis.

This work explores how the statistical method can be applied to quantitative reachability analysis. We propose a *statistical reachability analysis*, which tackles the quantitative reachability analysis problem with random sampling and statistical modeling. The main issue of statistical reachability analysis is how to estimate the reaching probability of a certain program state that has not yet been observed in the sampling process. To overcome this issue, we first suggest a naive approach of using two well-known estimators, Laplace smoothing and Good-Turing estimator [15], that can estimate the non-zero probability of unseen events from the

frequency of the seen events in a blackbox manner, i.e., without considering the structural aspect of the program. Then, we further investigate how the intrinsic structural property of the program, the dependency between program states, can be used to improve the estimation accuracy of the statistical reachability analysis, especially when the number of samples is small: the classic problem of sampling-based blackbox estimation. We claim that the structural relation between program states can be an essential ingredient to improving estimation accuracy. Consequently, we propose a *structure-aware estimator*, a novel statistical reachability estimator that considers the structural relation between program states.

To evaluate the performance of the statistical reachability analysis, we use a set of programs that have been used for evaluating the state-of-the-art probabilistic reachability estimator, and compare the statistical reachability estimator against two representative probabilistic reachability analysis, PSE [12] and PReach [26]. Our evaluation shows that the statistical reachability estimator can precisely estimate the reaching probability for all subject programs, mostly in less than 0.1 seconds, while probabilistic reachability analyzers fail or take a significantly longer time to estimate for a considerable amount of programs. In addition, we investigate the effect of using structural information on the estimation accuracy and the efficiency of the quantitative reachability analysis. We evaluate the structure-aware estimator against blackbox estimators with the Siemens suite and open-source softwares. Our empirical results show that, within 10% of the expected number of samples needed to reach the target state, the structure-aware estimator can estimate the reaching probability with less than one magnitude of error, while Laplace and Good-Turing estimator produces the estimates with 1.28 and 2.41 magnitudes of error, respectively.

The contributions of this paper are as follows:

- We propose statistical reachability analysis, a novel sampling-based quantitative reachability analysis that estimates the reaching probability of program states from a statistical perspective.
- Our empirical evaluation for a set of programs from SV-COMP benchmarks shows that the statistical reachability estimator can precisely estimate the reaching probability with small samples for all subjects. In contrast, probabilistic reachability analyzers applicablilty and performance are limited.
- We design a structure-aware reachability estimator that leverages the dependence information between program states to provide a more accurate estimation with a small number of samples.
- We evaluate the (blackbox and structure-aware) statistical reachability estimators on a large and complex real-world software.
- Our empirical results show that the structure-aware estimator is more accurate with a small number of samples than the blackbox estimators, indicating the advantage of using the structural information of the program.

2 BACKGROUND

2.1 Quantitative Reachability Analysis

Quantitative Reachability Analysis is an extension of reachability analysis that computes the probability of whether a certain program state is reached for arbitrary program execution. Given a program P, a set of program states S, and an arbitrary operational distribution of program execution E, the reaching probability $\Pr(s)$ of a program state $s \in S$ is defined as the probability of a random program execution $e \in E$ reaching the state s:

$$\Pr(s) = \sum_{e \in E} \Pr(e) \cdot \mathbb{1}(s \text{ is reached by } e), \tag{1}$$

where $\mathbb{1}(s \text{ is reached by } e)$ is an indicator function that returns 1 if s is reached by e and 0 otherwise.

Quantitative reachability analysis can be a primary goal of program analysis for figuring out the reaching probability of a certain critical program state. For example, it can provide how likely a crashing program state is reached in a normal workload of program execution. Such a crashing state could be an assertion that the developer placed in the program, assuming it is always true, or a recently discovered bug that is not yet fixed. It can also estimate for load testing a server-client program quantifying how often a certain query is sent to the server in various workloads in different stages of software deployment. Such an information could be useful for the service provider to decide the how many servers are needed to handle the workload. The result of the quantitative reachability analysis can further be used for the downstream software testing tasks. One can use it to design an optimal concolic testing strategy by balancing the cost for the concrete execution and the symbolic execution [28, 30].

2.2 Probabilistic Reachability Analysis

Probabilistic reachability analysis [11–13, 26], the primary approach for quantitative reachability analysis, has been widely studied in the last few decades. Based on the symbolic execution and the model counting [14], probabilistic reachability analysis computes the reaching probability of a program state directly from the source code. *Probabilistic Symbolic Execution (PSE)* [12] is a pioneering technique that computes the probability of executing portions of the program to perform a quantitative reasoning for symbolic execution. Given a path condition for executing a portion of a program, PSE computes the probability by counting the number of inputs that satisfy the condition using model counting. PSE solves the probabilistic reachability analysis problem by adding up all the probabilities of the paths that reach the target program state. Geldenhuys et al. show that PSE can estimate the probability of triggering a bug in the program with high accuracy. As the path explosion problem is a well-known obstacle for symbolic execution to analyze a practical program, several subsequent works went on to improve the scalability of PSE: Filieri et al. [11] proposed Statistical Symbolic Execution that still runs the model counting but only for subset of paths sampled from all existing paths and uses the information for Bayesian Estimation to estimate the gross probability. Gerrard et al. [13] proposed Conditional Quantitative Program Analysis, which blends evidence from multiple static analyses to reduce the search space of the input reaching a target program statement.

Most recently, PReach has been proposed to overcome the path explosion problem [26]. To avoid directly dealing with the probabilities of execution paths, PReach chooses to compute the probabilities of satisfying branch conditions in the program and use them to estimate the reaching probability of a program statement. Branch selectivity, which denotes the percentage of values satisfying the

branch condition, is calculated using model counting as the ratio of the number of values in the domain to the domain size of the variable type satisfying the branch condition. Then, PReach constructs a discrete-time Markov chain model representing the transition probabilities of the statement execution using the control flow graph together with the branch selectivity. The reaching probability of a program statement is computed by solving the Markov chain model. While PReach reduces the cost of the model counting, it can only approximate the reaching probability of a program statement since it does not consider the context of the program execution. The domain of the variable can be changed by the other statements executed before. Thus, the branch selectivity may significantly differ from the actual probability of satisfying the branch condition in the program execution. To overcome such an imprecision, PReach suggests applying abstract interpretation to infer the domain of the variable. There are two variations of PReach: PReach-I, which uses the interval domain, and PReach-P, which uses the polyhedra domain.

2.3 Statistical Estimator for Unseen Events

In statistics, while all observable events in the domain space have non-zero probability, some rare events may not be observed in the sample space. Estimating the probability of an unobserved event is the *sunrise problem*. To avoid assigning zero probability to unobserved events, several statistical estimators have been proposed.

Laplace smoothing, also known as additive smoothing, is one of the popular estimators for estimating the non-zero probability of an unobserved event by adding a small positive smoothing parameter α to all the events. The Laplace smoothing estimator estimates the probability of the i-th category as $\frac{c_i+\alpha}{n+\alpha d}$, where c_i is the number of i-th category samples, and $\alpha > 0$ is a smoothing parameter.

Another well-known estimator for estimating the probability of an unseen event is the *Good-Turing estimator* [15]. The Good-Turing estimator estimates the probability of discovering an object of hitherto unseen species from the number of already seen species. It first defines the incidence frequency Y_i for a species D_i as the number of times that the species i is observed in the sample O_n. Then, the number of singleton species f_1 is defined as the number of species that are observed only once, i.e., $|\{Y_i = 1\}_i|$. The Good-Turing estimator estimates the probability of discovering any unseen species in the next sampling as: f_1/n. The probability of discovering a certain unseen species D_i is $f_1/(f_0 n)$, where f_0 is the number of unseen species within O_n. However, in general, it is unknown how many species exist and, therefore, how many unseen species are in the sample O_n. Therefore, Good Turing provides an upper bound on the probability of discovering a certain unseen species D_i.

3 STATISTICAL REACHABILITY ANALYSIS

While probabilistic reachability analysis computes the true reaching probability of a program state,[1] one can *approximate* the reaching probability using stochastic sampling. Let us say we want to estimate the reaching probability $\Pr(s)$ of a program state s in a program P under a certain workload, where we can sample a random program execution from. Assuming we have sufficient number n of program execution samples O_n, the ratio of the number of samples that reach s to the total number of samples (i.e., empirical probability $\hat{\Pr}(s)$) will be a close approximation of $\Pr(s)$:

$$\hat{\Pr}(s) = \frac{X_s}{n} \xrightarrow{n\to\infty} \Pr(s), \tag{2}$$

where X_s is the number of executions in O_n that reach s.

However, for a certain program state that is extremely rare to observe, the empirical probability estimator can generate a substantial bias:

$$\mathbb{E}\left(\frac{X_s}{n} \mid X_s > 0\right) = \frac{\Pr(s)}{1-(1-\Pr(s))^n}, \quad \mathbb{E}\left(\frac{X_s}{n} \mid X_s = 0\right) = 0; \tag{3}$$

if the reaching probability $\Pr(s)$ is very small, the empirical probability becomes significantly larger than the true probability if it is observed (the left equation), and underestimated to zero if it is not observed (the right equation). Especially, such an underestimation can be detrimental in the program reliability analysis: the analysis may conclude that the program is reliable even though there is a reachable erroneous state in the program. Therefore, the statistical reachability analysis should always deal with the sunrise problem.

3.1 Blackbox Reachability Estimators

To estimate the reaching probability for both observed and unobserved program states, we employ the *Laplace estimator* and the *Good-Turing estimator*, which can deal with the sunrise problem. For given executions O and the target program state s, the *Laplace estimator* (*Lap*) estimates the probability of reaching s as:

$$Lap(s) = \frac{c_s + \alpha}{n + 2\alpha}, \tag{4}$$

where c_s is the number of executions that reach s, considering there are two categories: the executions that reach s and the executions that do not reach s. For the same setting, the *Good-Turing estimator* (*GoTu*) estimates the probability of reaching s as:

$$GoTu(s) = \begin{cases} c_s/n, & \text{if } c_s > 0, \\ f_1/n, & \text{otherwise,} \end{cases} \tag{5}$$

where f_1 is the number of singleton program states in o; it gives the empirical probability if the target program state is observed in the executions O; otherwise, it gives the Good-Turing estimation.

Drawbacks of Blackbox Estimators. A typical drawback of sampling-based estimators is that they may not be accurate for small sample sizes. For example, the theoretical background of the Laplace estimator lies in the normal approximation of binomial distribution, which is sensible if the number of samples is large. Similarly, Good also described that the estimation is expected to be good if the number of sample n is large and f_1 is not too small [15].

Another limitation of the above estimators is that they do not exploit the dependencies between the program states. For example, if a statement s_2 is the statement under the true branch of the if statement s_1, then the reaching probability $\Pr(s_2)$ is always smaller than (or equal to) $\Pr(s_1)$. However, if they are not reached in the sample executions, none of the above estimators can distinguish between the reaching probability of s_1 and s_2.

[1] While they aim to compute the true probability, often the analysis outcome is also an approximation due to theoretical limitations and impractical computational cost.

3.2 Structural Relation for Better Estimation

Program execution is a sequence of program state transitions, and, therefore, there exists a hierarchical dependence relation between the states. Often a rare event, such as a crash, of a program execution is placed deeply in the program's state hierarchy; they may occur only if several preconditions are satisfied during the execution. In such a case, the hierarchical relation between program states until reaching the rare event becomes a vital factor in estimating the probability of the rare event.

Structural relations between program states can be an indicator distinguishing the reachability probability of multiple unreached program states. Figure **??**, for instance, represents that the program state s_1 is the direct predecessor of s_2, similar to the if statement and its true branch we discussed in Section 3.1. Since s_2 can only be reached only if s_1 is reached, the reaching probability of s_2 is less than or equal to that of s_1.

Using the given executions and the structural information, we can even quantify the reaching probabilities of different unreached states. In Figure **??**, we assume that states s_1 and s_3 are reached 1,000 times and three times, respectively, from 1,000 executions, and those executions never reached their direct successor states s_2 and s_4. Then, we consider applying Laplace estimation on each s_1 and s_3 to estimate the reaching probability of s_2 and s_4:

$$\Pr(s_2) = \Pr(s_1) \times \frac{\alpha}{1,000 + 2 \times \alpha} \overset{\alpha=2}{=} 1 \times \frac{2}{1,004} \approx 0.0020,$$
$$\Pr(s_4) = \Pr(s_3) \times \frac{\alpha}{3 + 2 \times \alpha} \overset{\alpha=2}{=} 0.003 \times \frac{2}{10} = 0.0006.$$

The equation represents that there are more chances to reach s_2 by reaching s_1 than s_4 by reaching s_3, yet, if they reach their direct predecessors, s_4 is more likely to be reached than s_2.

Now, we can estimate the reaching probability of an unreached state, which exists deep in the structure, using the reaching probabilities of its predecessors and the path probability that lead to the unreached state. Given a structure and the executions, the state reached by the executions constructs the *frontier* of the reaching region. The frontier consists of states that have an unreached direct successor. For instance, the state R in Figure **??** belongs to the frontier. Then, there could be a hypothetical execution that can reach the unreached hidden state H from R following the path $R \rightarrow s_2 \rightarrow s_5 \rightarrow H$. We call this path *a hypothetical path* or *a hy-path* of H. The reaching probability of H by following the hypothetical path $R \rightarrow s_2 \rightarrow s_5 \rightarrow H$ is the product of the reaching probability of R and the transition probability at each state in the path. Assuming that there is no prior information on the unreached state, we may estimate the transition probability on each state as the reciprocal of the number of its direct successors. Then, the probability of executing the hypothetical path is

$$\Pr(R) \times \frac{\alpha}{\#(R) + 2 \times \alpha} \times \frac{1}{3} \times \frac{1}{2}.$$

Note that an unreached state can have multiple hy-paths. In general, we can estimate the reaching probability of an arbitrary unreached state h by summing up all the probability of executing the hypothetical path of h. Based on this motivation, we formally define the structure-aware reachability estimator in the next section.

3.3 Structure-aware Reachability Estimator

Notations & Hy-Path. Structure-aware reachability estimation is based on a given directed graph $G = (V, E)$, called a *structure graph*, representing the dependence relation between the program states in the program P. *Nodes* V is a set of discrete program states, and E is a set of *edges* representing the viable state transitions during the program execution. A typical example of such a graph can be a control-flow graph, a data-flow graph, or a call-graph, etc., which are widely used in program analysis, but, not limited to these, it can be any graph that *approximates* the dependency of the program states. We will further discuss the precision of the graph at the end of this section. A *path* $\mathbf{p} = (v_1, v_2, \ldots, v_n)$ in G is a sequence of nodes such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq n-1$. We call a path starting from an entry node (a node having no predecessor in G) and ending at the exit node (a node having no successor in G) a *complete path*, or an *execution* as a single program execution can be treated as sampling a complete path from G.

Given a set of executions $\boldsymbol{O}$, we partition V into two sets: a set of reached nodes V_r and a set of unreached nodes V_u. Then, the definition of a hy-path is as follows:

Definition 3.1 (Hy-Path (**hp**)). *Assume a node $h \in V$ is not reached for a given set of executions O; $h \in V_u$. A path* $\mathbf{hp} = \{p_1, p_2, \cdots, p_n = h\}$ *is a* hypothetical path *or a* hy-path *of h if $p_1 \in V_r$, $\{p_2, \cdots p_n\} \subset V_u$, and $\exists\, o \in \boldsymbol{O}$, whose execution is*

$$\mathbf{o} = \{entry = o_1, o_2, \cdots, o_n = p_1, \cdots, exit\},$$

such that

$$\mathbf{he} = \{entry = o_1, o_2, \cdots, o_n = p_1, p_2, \cdots, p_n = h, \cdots, exit\}$$

is a complete path. We call node p_1 a critical node *and complete path* **he** *a* hypothetical execution *of* **hp**.

Probability Estimation. Our estimator estimates the reaching probability of an unreached node $h \in V_u$ by *the sum of the reaching probability of all hy-paths of h*. The estimator theoretically over-approximates the reaching probability of h since every (hypothetical) execution reaching h contains at least one non-overlapping hy-path of h: for any hypothetical execution, one can consider the *last* reached node p_1 in the execution before reaching h. Then, the sub-path from p_1 to h is a hy-path of h. Now, the difference between the estimate and the estimand is the probability of the hypothetical execution that contains more than one non-overlapping hy-path of h. We claim that the bias is negligible in practice since the node h we want to estimate is already a hard-to-reach node for a large number of executions. Considering a hy-path instead of a whole (hypothetical) execution can significantly reduce the graph traversal space and thus the estimation time as the majority of reachable program states can be reached with a relatively small number of executions [5].

Path Probability. The key idea to estimate $\Pr(\mathbf{p})$, the probability of executing the path $\mathbf{p}$ in G, is to consider a path as the state transition of the *discrete-time Markov chain* defined on top of G. The probability of $\mathbf{hp} = \{p_1, p_2, \cdots, p_n = h\}$ is the reaching probability of node p_1 multiplied by the product of the transition probabilities

of all edges in the hy-path:

$$\Pr(\mathbf{hp}) = \Pr(p_1) \times \Pr(p_1 \rightarrow p_2) \times \prod_{i=2}^{n-1} \Pr(p_i \rightarrow p_{i+1}), \quad (6)$$

where $\Pr(x \rightarrow y)$ denote the state transition probability from state x to state y in the Markov chain. In **hp**, only the first node p_1 is reached by executions, and the rest of the nodes $p_2, \cdots, p_n$ are not reached. We use the empirical probability for estimating the probability of the first node p_1: $\hat{\Pr}(p_1) = |\{\mathbf{o} \in O \rightarrow p_1 \in \mathbf{o}\}|/|O|$. We estimate the transition probability $\Pr(p_2 \rightarrow p_1)$ using Laplace smoothing for the local executions that reach p_1: given that none of the n_1 number of executions that reach p_1 move to p_2, $\hat{\Pr}(p_2 \rightarrow p_1) = (n_1 + \alpha)/(n_1 + 2 \times \alpha)$. Then, for the remaining transition probability, we used the reciprocal of the number of children of the previous node as the transition probability: $\hat{\Pr}(p_{i+1} \rightarrow p_i) = 1/|p_i.\text{succ}|$.

Cycles. If the structure has cycles, there could be an infinite number of hy-paths of h. Such a cycle can be dealt with a stationary probability distribution calculation for the discrete-time Markov chain (e.g., the PageRank [7] or the random walk [8]), which is a well-known technique in the field of stochastic processes. In addition, based on the features of the given graph structure, we can further reduce the search space of the hy-paths, which will be discussed in Section 3.4 with a concretized analysis task for our evaluation.

Approximation of Structure Graph. It is worth noting that a structure-aware estimator can provide the reaching probability even if the given structure graph is abstracted or less precise compared to the graph with the exact dependence information of the program. Due to the dynamic nature of the program (e.g., dynamic function call), obtaining the exact dependence information between the program states is not always possible. Nonetheless, as the execution set size increases, the frontier of the reaching region gets closer to the target state, which means the distance between the critical node and the target state gets smaller. Thus, the bias due to the imprecision of the structure graph reduces, i.e., the estimation result is generally getting more accurate as the execution set size increases. Such a property of structure-aware estimation makes itself more robust to the imprecision of the given structure graph.

3.4 Optimization using the Structural Properties

We evaluate our structure-aware and blackbox reachability estimators on the *statement reachability estimation* problem, a typical problem for evaluating the reachability analysis methods [12, 26]. For the structure graph, we use the inter-procedural control-flow graph (InterCFG) as it represents the order of reaching the program statements during execution. We first set our assumption that the program has a single exit point while we argue about this assumption at the end of this section.

The property of the structure graph can be used to avoid unnecessary computation for the structure-aware estimator. In this section, we suggest three optimizations using the structural properties of the InterCFG: 1) terminating inter-procedural call, 2) modularizing the hy-path, and 3) loop handling.

Terminating Inter-procedural Call. A program execution may contain zero or more inter-procedural transitions. Given a path in InterCFG, one can identify the terminating inter-procedural call by matching the function-call edge and corresponding function-return edge in the path. Then, the whole sub-path under the terminating inter-procedural call can be replaced with a single edge connecting the calling node and the next node in the caller function; the transition probability of this edge is one because the execution will always return to the next node after the function call.

Modularizing the Hy-path. After removing the terminating inter-procedural calls, we can modularize the reduced hy-path into a sequence of intra-procedural paths, called *path units*; the probability of the hy-path becomes the product of the probabilities of the path units:

$$\Pr(\mathbf{hp}) = \prod_{\mathbf{pu} \subset \mathbf{rhp}} \Pr(\mathbf{pu}), \quad (7)$$

where **pu** denotes a path unit in a reduced hy-path **rhp**. The advantage of modularization is that the estimator can reuse the probability of the path units for multiple hy-paths. For example, if the function f, where the target statement s is located, has not been reached yet, every last path unit of the hy-path **hp** will be the same as $\mathbf{pu} = [\text{entry} \rightarrow \cdots \rightarrow s]_f$, where $[\cdots]_{func}$ indicates the path unit belongs to the function *func*. Therefore, until the function f is reached, the probability of the path unit **pu** can be efficiently reused. Our empirical evaluation shows that modularization can significantly reduce the estimation time.

Loop Handling. The cycle in an inter-procedural path occurs either by a loop appearing at a path unit-level or a cyclic call sequence appearing at a modularized hy-path-level. Using the feature of the inter-procedural control-flow graph, we can avoid computing the stationary probability distribution of the Markov process to deal with the cycles in the hy-path.

When estimating the probability of a path unit, the structure-aware estimator can only consider the acyclic path unit by adjusting the transition probability so that the probability of an acyclic path includes the chances of the cyclic path. There are two cases: 1) the path unit passes a loop (it does not end in the loop), 2) the path unit ends in the loop. If a path unit passes a loop, the adjustment to the transition probability $\Pr(\text{loop-entry} \rightarrow \text{loop-exit}) = 1$ since every loop entrance must reach the loop exit if there is a single program exit. Assume the path unit's last node e is in the loop, and let us denote the true branch of the loop a loop-start. The original transition probability $\Pr(\text{loop-start} \rightarrow \text{loop-entry})$ before adjustment is $1/|\text{loop-entry.succ}| = 1/2$. Let us say the reaching probability of e after entering the loop is q; $\Pr(\text{loop-start} \rightarrow^+ e) = q$.[2] Then, the reaching probability of e in any loop iteration after reaching the loop entry is:

$$\frac{q}{2} + \frac{1-q}{2} \cdot \frac{q}{2} + \left(\frac{1-q}{2}\right)^2 \cdot \frac{q}{2} + \cdots = \frac{q}{1+q} = q \times \frac{1}{1+q}. \quad (8)$$

Thus, the probability of a path unit can be estimated by 1) starting from the last node of the path unit, 2) multiplying the transition probability in the reverse direction of the path unit, and 3) if it meets the loop entry, adjust the transition probability to $\Pr(\text{loop-entry} \rightarrow \text{loop-start}) = \frac{1}{1+q}$, where q is the probability computed until the current process. As the cyclic call sequence works similarly in the

[2] $\Pr(x \rightarrow^+ y)$ is the sum of the probabilities of the intra-procedural paths from x to y

Table 1: RQ1 subject programs from SV-COMP [25].

jpf-regress. (26)	ExMIT-T, Exe1-F, Exe2-F, Exe4-F, Exe6-F, Exe8-F, Exe10-F, Exe10-T, Exe12-F, Exe12-T, Exe13-T, Exe14-T, Exe15-T, Exe18-F, Exe19-T, Exe20-F, Exe20-T, Exe26-F, Exe27-F, FNEG-T, LCMP-T, Simple-F, Simple-T, Suzette-F, Suzette-T, Assign-T
jbmc-regress. (4)	assert3, if_icmp1, switch1, Token2
algorithms (2)	InsertSort2, RBTree1

path-level, we can make the same adjustment to avoid computing the stationary probability distribution of the Markov process.

We finally mention the single exit assumption. In the general case, a program can have multiple exits. When there is a program exit between the frontier of the reaching program statements and the target statement, our estimator may overestimate the reaching probability of the target statement. The bias due to the imprecision of the single exit assumption will decrease as the sample size increases since there will be less chance of having a program exit between the frontier and the target statement.

4 EXPERIMENTAL SETUP

4.1 Research Questions

We mainly ask two research questions to evaluate the performance of the statistical reachability analysis.

RQ1: How does the statistical reachability analysis perform compared to the probabilistic reachability analysis? In this research question, we investigate the limitation of existing analytic methods and how the statistical reachability analysis performs compared to the probabilistic reachability analysis. Our investigation considers both the accuracy of the estimation for the program with nontrivial semantics, and the scalability of the method. We use PSE and PReach as baseline probabilistic reachability analyzers for comparison with the statistical reachability estimators on the benchmark programs used in the PReach work.

RQ2: How do the structure-aware and the blackbox reachability estimators perform on hard-to-reach states in a complex program? We claim that the statistical reachability estimators are scalable regardless of how complex the program semantic is. To verify this claim, we investigate the performance of the statistical reachability estimators for estimating the reaching probability of hard-to-reach states in the program with bigger sizes and much more complex semantics than the benchmark programs used in **RQ1**. We also investigate the efficiency of the structure-aware model compared to the blackbox model.

4.2 Subject Programs and Target Statements

To evaluate the performance of statistical reachability estimators compared to probabilistic reachability analyzers, we concretize the problem of quantitative reachability analysis into the problem of *statement reachability analysis* and use the same benchmark programs used in the PReach work [25]: 142 java programs from Competition on Software Verification 2021 (SV-COMP). The target statements are the assertions in the programs. Most 142 programs have very primitive semantics to reach the assertion; a single comparison ($<, >, ==, !=$) to an input is the only condition for the control flow to the assertion. As our interest is in more realistic programs, we filtered out those programs with primitive semantics and left only the programs whose semantics have meaningful changes to the reaching probability. Our selection criterion is that the program semantics should update the domain of the value used in the comparison for reaching the assertion and affecting the branch probability. We manually investigate all 142 programs and select 32 programs after filtering. Table 1 shows the selected programs.[3] The average non-comment-line-of-code (NCLOC) is 35.2.

Table 2: Statistics of RQ2 subject programs from Siemens suite (above five) and the real-world programs (below five)

Program	NCLOC	# Func	# BB	GT
tcas	146	9	63	5.37E-04
schedule2	332	17	138	3.99E-04
totinfo	349	7	132	9.2E-04
printtokens2	438	19	198	7.82E-03
replace	534	21	228	2.73E-04
gif2png*	988	27	700	2.95E-04
jsoncpp	7,251	1,328	5,938	2.28E-03
jasper*	17,385	720	14,417	2.48E-04
readelf	22,347	477	18,578	1.99E-07
freetype2	44,686	1,635	27,521	8.25E-08

Since the program size is relatively small, there is no singleton after a few iterations of the sampling process. Therefore, we only use the Laplace estimator for the statistical reachability analysis method in **RQ1**. To get the ground-truth reaching probability of the assertion, we check the semantics of each program and manually compute the reaching probability; we consider the same domain (a signed 31-bit for an integer input and a length of 16 with all printable ASCII characters for a string input) and assuming the uniform distribution of the input domain as the PReach work. To validate the ground-truth, we seperately run a sufficiently large number of iterations of the sampling process and compare the ground-truth with the empirical reaching probability.

Subjects used in **RQ1** are relatively small-sized with less complexity. To evaluate the performance of statistical reachability estimators in realistic programs, we choose five middle-sized programs from the Siemens suite [9][4] and several large-sized free and open-source (FOS) C/C++ applications and libraries. Table 2 shows the selected programs, number of lines, functions, and basic blocks.

To remove the selective bias as well as to evaluate estimators with a sufficiently challenging task, we select the statement that is most-frequently lastly-found while running multiple greybox fuzzing on each subject program. For the Siemens suite subjects, we run greybox fuzzing for six minutes with 990 repetitions and record the statement coverage hit-counts at each second. For the FOS subjects, we run ten repetitions of one-week greybox fuzzing taking more than one CPU year and record the statement coverage

[3]The program names are abstracted for the space issue.
[4]we remove 'schedule' and 'printtokens' as they have the same semantics with 'schedule2' and 'printtokens2' but different implementation

hit-counts at each 15 minutes. If the fuzzing finds the crashing input during the campaign for FOS subject, we use the last statement of the crashing path as the target statement considering the practical reliability analysis scenario.[5]

4.3 Sampling Process

We use the greybox fuzzing data to generate random samples for the statistical reachability analysis. Instead of using the raw statement coverage hit-count from the fuzzing campaign, which contains the adaptive bias due to the guidance of the greybox manner, we create a simulated random sampling that is representative of the average behavior of multiple greybox fuzzing campaigns. The simulated random sampling process is as follows: for each greybox fuzzing campaign that discovers the target statement in its campaign, we record the hit-count of the statement coverage at the timestamp where the target statement is reached first. The hit-count data divided by the number of samples until the timestamp (i.e., empirical probability) becomes the expected coverage per sample of the certain fuzzing campaign. By averaging the expected coverage per sample for all the campaigns, we get the average expected coverage per sample of the simulated random sampling. We use the target statement's ground-truth reaching probability, GT, as the empirical probability of this average expected coverage, which is presented in Table 2. Then, we can simulate the hit-count data for an arbitrary number of samples by multiplying the average expected coverage per sample by the number of samples. For example, the hit-count of a statement whose average expected coverage per sample is 0.372 with 100 samples is $\lfloor 0.372 \times 100 \rfloor = 37$. The implementation of the simulated random sampling can be found in the public repository. For **RQ2** evaluation, we generate 10% of the samples expected to reach the target statement: $\mathrm{GT}^{-1}/10$ per subject program.

There are two main advantages of using simulated random sampling. First, it can be a representative of the average behavior of multiple fuzzing campaigns. Different fuzzing campaigns can have different hitting behavior due to randomness. Such a randomness highly affects the Good-Turing estimator, which uses the number of singletons, and the structure-aware estimator, which considers what the critical node is and how many times it is hit. Therefore, representative of the average behavior is needed to remove the random bias for the evaluation. Second, random sampling from an expected coverage per execution can simulate the hit-count data for an arbitrary sample size. Since recording every hit-count data for each new sample for a long fuzzing campaign of large programs is not feasible, the hit-count data for our five open-source programs are only available every 15 minutes. Therefore, the data is coarse-grained as the total number of data points is limited, which deteriorates the statistical reachability analysis evaluation. Conversely, hit-count data for an arbitrary sample size can provide a fine-grained evaluation of the statistical reachability analysis.

4.4 Evaluation Metric & Environment

For **RQ1**, we record the *wall-clock time taken* to measure the scalability of the methods. The wall-clock time taken for the statistical reachability estimator also includes the time for the sampling process, as the computing cost of Laplace estimator is practically negligible. While it is clear that the accuracy of the statistical reachability estimator would increase as the sample size increases, we define a threshold of the successful estimation to measure how much cost is needed to get an accurate estimation. We say the statistical reachability estimator succeeds in the estimation with N_s executions if all the errors of ten continuing estimates until N_s-th estimation (e.g., $Lap(s, N_s - 9), \cdots, Lap(s, N_s)$ for Laplace estimator) fall between 99% and 101% of the ground-truth probability in logarithmic scale.

$$0.99 \times \left|\log_{10}(GT)\right| < \left|\log_{10}(e)\right| < 1.01 \times \left|\log_{10}(GT)\right|,$$

where e is the estimated probability and GT is the ground-truth probability. We use logarithmic scale since it is more sensitive to the estimation error when the probability is small. The wall-clock time taken for the statistical reachability estimator in **RQ1** is then the time taken until the N_s-th estimation. Our timeout threshold for the estimation is 15 minutes. If the estimation does not finish within 15 minutes, we consider the estimation as failed.

For **RQ2**, we compute the accuracy of the estimated probability by computing the logarithmic error between the estimated probability and the ground-truth probability:

$$\text{log-err}(e, GT) = |\log_{10}(e) - \log_{10}(GT)|.$$

If the log-err is 1, it means that the estimated probability is one order of magnitude different from the ground-truth probability. We measure the time taken for both the sampling process and estimation in **RQ2**. In addition, we also record the time taken of the estimation for the structure-aware estimator explicitly to compare the scalability of the structure-aware estimator itself with the black-box estimator whose time taken for the estimation is negligible. For all the experiments, including the greybox fuzzing campaigns and the estimation process, we repeat the process ten times and report the average value to reduce the random bias.

For the experiments, we use the PSE implementation in [21], which is implemented on top of Symbolic PathFinder (SPF) [24] as the original work does, and uses LattE [27] model counting tool. Since the implementation does not support a signed 32-bit domain, we use a signed 31-bit domain for PSE. For the Laplace and the structure-aware estimator, we use the smoothing factor $\alpha = 2$ as it is the representative value driven from the mid-point of the Wilson interval with 95% confidence [29]. We use PReach implementation provided by the authors [25] with the polyhedra domain as the default abstract interpretation domain (PReach-P). All the experiments are performed in a Ubuntu 18.04 docker container with 64 cores of AMD EPYC 7713P @ 2.0GHz and 251GB memory.

5 RESULT

5.1 RQ1: Analytic Method vs. Statistical Method

Quantitative Results. Table 3 shows the quantitative reachability estimation result for the programs we select from the SV-COMP benchmark. The first and second columns show the program name and the ground truth reaching probability of the target assertions. Each (3rd, 4th), (5th, 6th), and (7th, 8th) column represents the result from PSE, PReach, and Laplace estimator. Esti(·) and T(·) are estimator's probability estimate and time spent.

[5] '*' next to the subject name indicates the target node of the subject is the final statement of the crashing path.

Table 3: Quantitative reachability estimation for SV-COMP 2021 benchmarks. Esti(·) and T(·) are the probability estimate and the time spent for the estimator; (TO, NL, DTMC) are the failure states of the estimation. O/X in the parenthesis after the value represents whether the estimator succeeds.

Program	GT	Esti(PSE)	T(PSE)	Esti(PR)	T(PR)	Esti(Lap)	T(Lap)
ExMIT-T	~0	4.7E-10 (O)	.866s	7.6E-06 (O)	14.9s	1.0E-06 (O)	0.044s
Exe1-F	0.49	NL (X)	-	0.500 (O)	13.5s	0.489 (O)	0.006s
Exe2-F	0.2	NL (X)	-	0.125 (X)	14.6s	0.199 (O)	0.003s
Exe4-F	0.25	NL (X)	-	0.125 (X)	14.7s	0.248 (O)	0.014s
Exe6-F	1.0	NL (X)	-	2.3E-10 (X)	14.8s	0.990 (O)	0.001s
Exe8-F	0.3	NL (X)	-	0.500 (X)	14.7s	0.300 (O)	0.005s
Exe10-F	0.25	NL (X)	-	0.250 (O)	14.5s	0.250 (O)	0.005s
Exe10-T	~0	NL (X)	-	1.2E-10 (O)	14.5s	1.0E-06 (O)	0.085s
Exe12-F	0.5	0.500 (O)	.934s	0.500 (O)	14.6s	0.501 (O)	0.004s
Exe12-T	0.375	0.250 (X)	.966s	0.375 (O)	14.6s	0.376 (O)	0.007s
Exe13-T	~0	0 (O)	.909s	5.0E-11 (O)	13.7s	1.0E-06 (O)	0.087s
Exe14-T	0.25	0.5 (X)	.860s	0.25 (O)	11.9s	0.251 (O)	0.018s
Exe15-T	0.25	0.125 (X)	.910s	0.25 (O)	13.1s	0.251 (O)	0.011s
Exe18-F	0.5	NL (X)	-	0.500 (O)	14.5s	0.502 (O)	0.011s
Exe19-T	0.25	0.375 (X)	.950s	0.245 (O)	14.5s	0.251 (O)	0.015s
Exe20-F	0.25	NL (X)	-	0.125 (X)	13.6s	0.249 (O)	0.008s
Exe20-T	0.5	0.500 (O)	.903s	0.5 (O)	14.5s	0.500 (O)	0.008s
Exe26-F	0.5	NL (X)	-	0.245 (X)	14.7s	0.500 (O)	0.006s
Exe27-F	0.5	0.500 (O)	.849s	0.500 (O)	14.7s	0.500 (O)	0.004s
FNEG-T	0	0 (O)	.850s	0.25 (X)	14.5s	1.0E-06 (O)	0.045s
LCMP-T	0	0 (O)	.832s	0.5 (X)	14.9s	1.0E-06 (O)	0.044s
Simple-F	0	0 (O)	.854s	TO (X)	-	1.0E-06 (O)	0.048s
Simple-T	0	0 (O)	.844s	TO (X)	-	1.0E-06 (O)	0.047s
Suzette-F	0.25	0.250 (O)	.910s	4.7E-10 (X)	13.8s	0.249 (O)	0.030s
Suzette-T	~0	2.6E-9 (O)	.926s	2.6E-09 (O)	14.4s	1.0E-06 (O)	0.084s
Assign-T	0	0 (O)	.841s	0.25 (X)	14.6s	1.0E-06 (O)	0.045s
InsertSort2	2.1E-02	TO (X)	-	2.5E-11 (X)	15.8s	2.1E-02 (O)	4,904s
RBTree1	0.125	TO (X)	-	DTMC (X)	14.4s	0.124 (O)	0.002s
assert3	~0	4.7E-10 (O)	.847s	2.3E-10 (O)	10.6s	1.0E-06 (O)	0.044s
if_icmp1	0	0 (O)	.856s	5.0E-11 (O)	10.5s	1.0E-06 (O)	0.045s
switch1	~0	2.8-09 (O)	1.03s	0.0 (O)	11.9s	1.0E-06 (O)	0.044s
Token2	4.8E-04	NL (X)	-	TO (X)	-	5.2E-04 (O)	0.545s

Among 32 subject programs, PSE successfully estimates the reaching probability of 15 programs. The symbolic expression of the path condition to reach the target assertion contains non-linear terms in 11 programs (NL in the table), which is not supported by PSE implementation. PSE timeouts (TO in the table) for the two programs, which are the exact two programs containing a loop. PSE estimates the wrong probability for the other four programs because the implementation does not consider the overflow/underflow of the integer variables. On average, PSE spends ~1 second on the estimation. In the case of PReach, the estimation succeeds only for 17 out of 32 programs. For 15 failing estimations, there is one timeout during the abstract interpretation, two timeouts during the model counting (TO in the table), and one divergence of DTMC (DTMC). For the remainings, PReach estimated far smaller/larger probabilities than the ground truth. Without the cases where the estimation fails, the time spent for estimation is over ten seconds.

On the other hand, the Laplace estimator successfully estimates the reaching probability for all 32 programs. Except **InsertSort**, the time spent is strictly less than one second, including the time for the sampling (program executions). Among the 31 programs, the average time spent estimating the assertion statements with feasible reaching probabilities (GT> 10^{-6}) is 0.039 seconds (median: 0.007), and the average number of samples needed (N_s) is 9,615 (median: 1,531). To achieve an estimated probability of 10^{-6} for the statements infeasible to reach, the Laplace estimator requires 2×10^6 samples, and the average time spent for it is 0.055 seconds

```
1 void test(String line) {
2   String[] toks =
        line.split(" ");
3   int i = 0;
4   for (String t : toks) {
5     if (i == 3)
6       assert false;
7     ++i;
8   }
9 }
```

(a) TokenTest02

```
1 void test(int x, int z) {
2   if(z < 0) return;
3   // instead of int y = 3;
4   int y = call("./ret3.sh");
5   z = x - y - 4;
6   if (x < z)
7     assert false;
8   else
9     print("b4");
10 }
```

(b) Exe13-T

```
1 void test(int i) {
2   if (i >= 1000)
3     if (!(i > 1000))
4       assert false;
5 }
```

(c) assert3

```
1 void test(int z) {
2   z = z % 5 - 2;
3   if (z < 0) print("b1");
4   else assert false;
5 }
```

(d) Exe8-F

Figure 1: Simplified pseudocodes of RQ1 subjects.

(median: 0.045). Only **InsertSort2** takes 4,904 seconds, mostly on the sampling process, not the estimation. This is because **InsertSort** runs the insertion sort in the worst-case scenario ($O(n^2)$) for a random positive integer number length ($[1, 2^{31} - 1]$) array.

> Our result shows that the Laplace estimator can successfully estimate the reaching probability of all subjects in Table 3 with high precision, generally, in a short period of time. On the other hand, PSE and Preach fail to estimate the accurate reaching probability of nearly half of the subjects.

Qualitative Analysis. We further investigate the properties of the programs that prevent the analytic approach from estimating the correct reaching probability of the program state.

Token2 (Figure 1a) epitomizes the limitation of the previously proposed probabilistic reachability analysis. An arbitrary size of the array significantly increases the domain space (Line 1), whose complexity becomes squared after the String API `split` is applied (Line 2). It is non-trivial and requires manual effort to interpret the semantics of any API call (Line 2). The loop in Lines 4-9 are a typical example of the path explosion problem; the number of paths to consider would grow exponentially with the size of the array if the true branch were not terminating the loop. The non-deterministic loop iterations (Line 4) also obstructs the scalability/precision of the analysis as it needs to consider the maximum number of iterations. Finally, the domain of variable `i` in Line 6 keeps changing at each loop iteration, which makes branch selectivity-based analysis difficult to compute the correct probability. According to the result, PSE fails to estimate the reaching probability for **Token2** due to the limited support for the String API. Even if the String API is supported, the path explosion problem would still make the analysis hardly feasible, as we have seen for the programs **InsertSort2** and **RBTree1** in Table 3. Both of the abstract interpretation (interval and polyhedra) of PReach reaches the timeout limit for **Token2**. The ground truth reaching probability of **Token2** is

$$1 - \left(\frac{94}{95}\right)^{15} - {}_{15}C_1 \frac{1}{95}\left(\frac{94}{95}\right)^{14} - {}_{15}C_2 \left(\frac{1}{95}\right)^2 \left(\frac{94}{95}\right)^{13} = 4.8267e-04,$$

which is the probability of the input string having more than two spaces. The vanilla PReach without the abstract interpretation fails to estimate ($\hat{\Pr} = 0.333$) the correct reaching probability to the assertion. Conversely, the Laplace estimator successfully estimates the probability with less than 1% of log-scale error in a half-second.

By computing the branch selectivity probability and inducing the path probability using DTMC, PReach avoids the path explosion problem. However, applying the model counting to each branch ignores the domain change of variables during execution, which may lead to a significant inaccuracy in the probability estimation. For instance, without abstract interpretation, PReach computes the reaching probability of the assertion in **assert3** (Figure 1c) as $0.25 = 1/2 \times 1/2$, where $1/2$ stands for the branch selectivity for each branch, yet the true probability is $1/2^{32}$. Abstract interpretation, the solution by PReach, can partially solve the problem of domain change. While it can assist the program like **assert3**, which still has a uniform distribution after the domain change, it fails to handle the case of **Exe8-F** (Figure 1d), where the domain space becomes non-uniform; the value distribution of variable `z` at Line 3 in Figure 1d is a non-uniform distribution between `[-6, 2]`, where -2 has a double probability (2/10) than other values (1/10) due to the previous instructions. Therefore, while the true probability of reaching the assertion statement at line 9 is $3/10 = 0.3$, the model counting method computes $3/9 = 0.333$, ignoring the non-uniform distribution of the domain space.[6] In addition, a disjoint domain space or a domain space that requires a complex domain abstraction exists in the real world, which also hinders the analytic approach.

Finally, we mention the opaque code problem. By simply changing the expression '3' on the righthand side of Line 3 in **Exe13-T** (Figure 1b) to an external call to a shell script that returns 3, so that the semantics of the program remains unchanged, PSE and PReach fail to determine the infeasibility to reach the assertion, returning 0.25 of reaching probability, because the code is unavailable. For the same program with the shell script, the Laplace estimator correctly estimates that the assertion statement is infeasible to reach.

Our qualitative investigation illustrates the limitation of the probabilistic reachability analysis to scale to real-world programs. On the other hand, the statistical reachability analysis can scale independently of the complexity of the program semantics, as it requires only the samples of program executions.

5.2 RQ2: Evaluation of Structural Information

Figure 2a shows the estimation results of the three statistical reachability estimators on the programs from the Siemens suite and FOS C/C++ programs/libraries. The upper side of the figure shows the raw estimated reaching probability of the target statements, and the below side shows the log-err compared to the ground-truth reaching probability. For all the subject programs, the result shows that the structure-aware estimator is the most accurate, and the Good-Turing estimator is the least accurate across the different sample size; the structure-aware estimator is always better than or equal to the Laplace estimator and so is to the Good-Turing estimator except for very few cases due to the unprecise estimation with a very small samples. Due to the non-existence of the singleton, the Good-Turing estimator produces no estimation for some sample size. For instance, it does not produce any estimation for the 'Printtokens2' and 'Jsoncpp.'

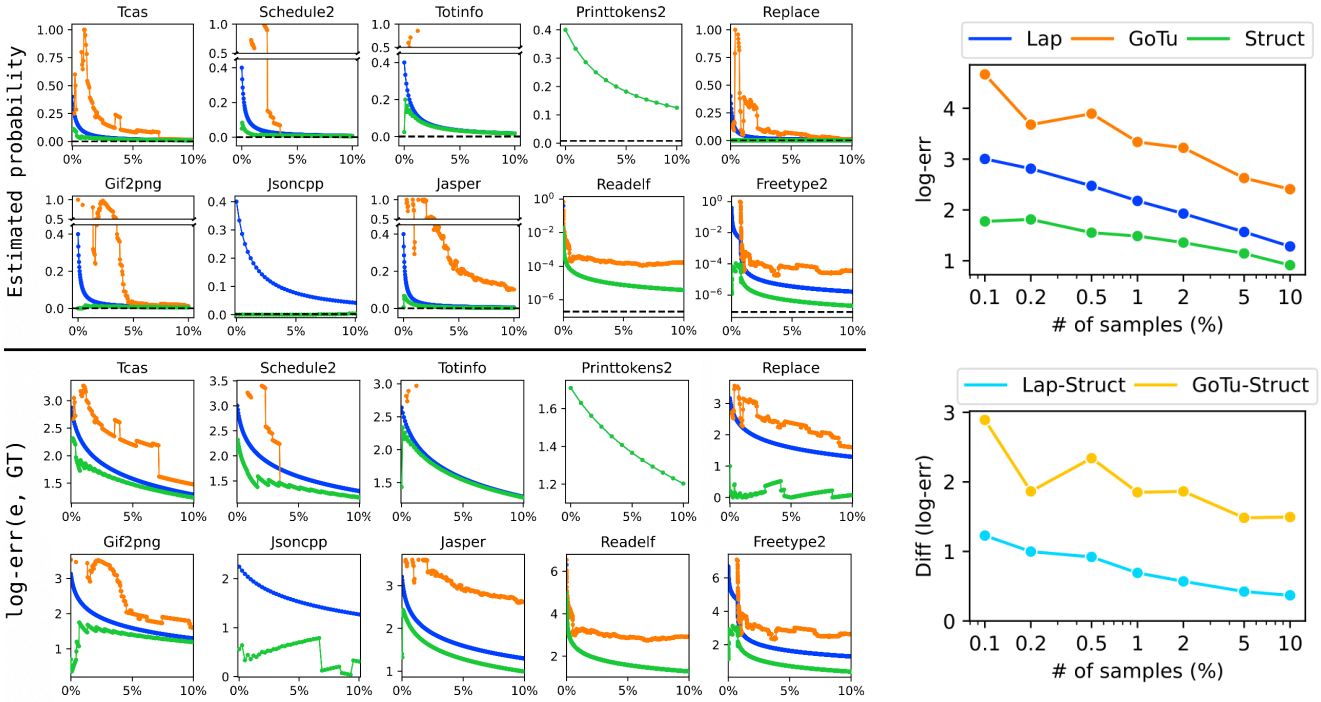
Figure 2b presents the average log-err of the three statistical reachability estimators across the programs in the log-scale. In general, the error of all three estimators decreases as the number of samples increases. Each Laplace, Good-Turing, and structure-aware estimator has the log-error of 3.00, 4.67, and 1.77, respectively, when 0.1% of the expected number of samples needed to reach the target statement is provided. Only the structure-aware estimator reaches a log-error below 1 (0.91), while Laplace and Good-Turing estimators get to 1.28 and 2.41, respectively. The below side of Figure 2b presents the difference in the average log-err between the structure-aware estimator and the other two estimators. The maximum difference between Laplace (Good-Turing) and structure-aware estimator until 10% of the samples is 1.99 (4.08), and the difference decreases as the number of samples increases.

It is worth noting that the structure-aware estimator is also cost-efficient. Table 4 shows the average time spent by the structure-aware estimator across the programs; on average, it took 0.012 seconds for the estimation (median: 5e-04s), which is comparable to the two other estimators, whose computation formula is rudimentary. The last three columns of Table 4 show the total (sampling + estimation) time spent by the three estimators *to reach* the estimated probability of log-err$(e, GT) < 2$ within 10% of the executions, where the bold values denotes the least time among the three estimators. 'To reach' means that *every* estimate after the shown time is within the error bound. The structure-aware estimator requires the least time for most programs. In contrast, the Good-Turing estimator could not produce the estimate within the error bound using 10% of the executions for several programs. Since the structure-aware estimator gives a more accurate estimation given the same number of samples, often the overall time spent by the structure-aware estimator is less than the other two estimators.

Table 4: The average estimation time and the total time to produce the estimated probability e of log-err$(e, GT) < 2$. '-' denotes that the estimator cannot produce the estimated probability of log-err$(e, GT) < 2$ within 10% of the samples.

Program	Esti(Struct)	Total(Lap)	Total(GoTu)	Total(Struct)
tcas	2.14e-04s	1.53e-01s	6.04e-01s	**3.63e-02s**
schedule2	2.43e-04s	2.54e-01s	4.75e-01s	**4.88e-02s**
totinfo	9.20e-04s	1.18e-01s	-	**9.30e-02s**
printtokens2	1.28e-04s	**4.54e-03s**	-	4.67e-03s
replace	2.74e-04s	3.07e-01s	1.32e+00s	**4.65e-03s**
gif2png	6.06e-03s	4.03e-01s	1.07e+00s	**1.24e-02s**
jsoncpp	7.30e-04s	1.83e-02s	-	**4.39e-03s**
jasper	2.48e-04s	3.37e-01s	-	**1.40e-01s**
readelf	8.56e-02s	**4.82e+02s**	-	4.83e+02s
freetype2	2.77e-02s	7.04e+02s	-	**4.41e+00s**

[6] In Table 3, PReach-P estimated 0.5 due to miss managing the '%' operator.
[7] The horizontal dashed lines in the estimation plots represent the ground-truth reaching probability of the target statements. The y-axis scale of 'Readelf' and 'Freetype2' estimation plots is logarithmic.



(a) Individual plots for each subject[7] (y: # of samples (%))

(b) Average log-err and differences

Figure 2: Three statistical reachability estimators' (Good-Turing: orange, Laplace: blue, Structure-aware: green) reaching probability estimation and the error of target statements in the programs from Siemens suite and FOS C/C++ programs/libraries.

The statistical reachability estimators are able to estimate the reaching probability for practical-sized programs. Our structure-aware estimator is the most accurate among the three estimators: within 10% of the expected number of executions needed to reach the target statement, the the estimate is less than one order of magnitude away from the ground-truth probability.

6 THREATS TO VALIDITY

Various threats to the validity need to be concerned about as we evaluate the performance of the reachability estimators empirically. *External validity* concerns whether the results from the study can be generalized. To mitigate this concern, we use various programs with different sizes and complexity: SV-COMP 2021, Siemens suite, and FOS software. In **RQ1**, we use the same benchmark used by the former study for a fair comparison and choose the subjects based on the selection criteria. We extend our investigation to the programs with a larger size and complexity in **RQ2** for a more general evaluation. *Internal validity* concerns the degree of confidence of our study, having not been influenced by any factor beyond the scope of the study. First, to mitigate the randomness of the experiment, we use numerous repetitions of greybox fuzzing campaigns and run every estimator with ten repetitions. Second, to avoid missing any potential error in our evaluation and to facilitate the reproduction of our study, we make our scripts and data publicly available.

7 RELATED WORK

Beyond the quantitative reachability analysis, Böhme and colleagues study the empirical method and the statistical framework for program analysis in general [4, 23] and for software testing specifically [3, 5, 6]. For instance, they applied the statistical approach to analyze various properties of blackbox fuzzing campaigns, including the total feasible branch coverage, the additional time required to cover 10% more branches, and the residual risk that a vulnerability exists when no vulnerability has been discovered so far. Later, they extended the approach to greybox fuzzing, where the sampling distribution is updated during the testing process, suggesting a methodology to avoid the adaptive bias in the statistical model [6].

Another line of work that exploits the observation-based statistical approach to estimating the program properties has been explored in the program dependency analysis [17]. Binkley et al. first proposed an Observation-based Slicing (ORBS) [1], a program slicing technique that can be applied to a program written in multiple programming languages. It has shown that, regardless of how actually the program is implemented, if the runtime behavior of a program state is changed after perturbing another program state, there is a dependency between the two states. Using this observation, ORBS can approximate the program dependency even for the dependency that cannot be captured by the static analysis [2]. Later, Lee et al. adopted statistical modeling and extended this slicing technique to a general program dependency analysis technique [18, 19].

8 DISCUSSION & FUTURE WORK

Throughout the evaluation, we have shown that the statistical reachability analysis is a promising approach to estimating the reaching probability of a program. Not only is it scalable, but it can also provide probability estimation with high precision for programs that are not amenable to the probabilistic reachability analysis. Furthermore, our novel structure-aware estimator demonstrates the benefit of incorporating structural information into the statistical reachability analysis. The structure-aware estimator can provide a more accurate estimation even with a small number of execution samples.

The main advantage of our structure-aware reachability estimator is that it systematically considers the hierarchical relation between the reached and unreached states. Therefore, the estimation is well-grounded in terms of the given structure (as we illustrated in Figure **??**) and expected to be more accurate than the existing statistical estimators that consider the program as a blackbox. At the same time, the structure-aware reachability estimator is much more scalable than the probabilistic reachability analysis. Our estimator can employ any structural information that can be driven from a light-weighted static analysis, which is more applicable than the symbolic execution and the model counting. For the future work, we will investigate how those advantages of statistical/structure-aware reachability analysis can actually benefit the concrete downstream applications on software testing and maintenance.

The probability distribution of reaching the successor program states from the unreached program states has been approximated by the uniform distribution in our structure-aware estimator. While this approximation is reasonable when there is no prior knowledge, any prior knowledge can be integrated into the structure-aware estimator by modifying the probability distribution. For instance, the branch selectivity from PReach can be used for the probability distribution of reaching the successor program states. Employing the information from the static analysis can also be helpful to improve the precision/soundness of the estimation result. For instance, some statistical estimator (e.g., Laplace) may assign nonzero probability to the *unreachable* program states, which can be avoided by using the information from the static analysis. Nonetheless, we avoid using the model counting-based methods in this study since it still requires considerable computational effort to run the model counting for every branch in a large program, and it will restrict the domain of the variable to the integer domain. In the future, we will investigate the beneficial integration of analytic and statistical methods.

9 DATA AVAILABILITY

All data & scripts are publicly available through Zenodo: https://doi.org/10.5281/zenodo.7612964.

ACKNOWLEDGEMENTS

This work was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State Governments - EXC 2092 CASA - 390781972.

REFERENCES

[1] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. 2014. ORBS: Language-independent Program Slicing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 109–120. https://doi.org/10.1145/2635868.2635893

[2] D. Binkley, N. Gold, M. Harman, S. Islam, J. Krinke, and S. Yoo. 2015. ORBS and the limits of static slicing. In *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 1–10. https://doi.org/10.1109/SCAM.2015.7335396

[3] Marcel Böhme. 2018. STADS: Software Testing as Species Discovery. *ACM Transactions on Software Engineering and Methodology* 27, 2, Article 7 (June 2018), 52 pages. https://doi.org/10.1145/3210309

[4] Marcel Böhme. 2022. Statistical Reasoning About Programs. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, USA) *(ICSE 2022)*. 5 pages. https://doi.org/10.1145/3510455.3512796

[5] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the Exponential Cost of Vulnerability Discovery. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 713–724. https://doi.org/10.1145/3368089.3409729

[6] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholz. 2021. Estimating Residual Risk in Greybox Fuzzing. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 230–241. https://doi.org/10.1145/3468264.3468570

[7] Sergey Brin and Lawrence Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems* 30, 1-7 (1998), 107–117.

[8] Michael B. Cohen, Jonathan Kelner, John Peebles, Richard Peng, Aaron Sidford, and Adrian Vladu. 2016. Faster Algorithms for Computing the Stationary Distribution, Simulating Random Walks, and More. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*. 583–592. https://doi.org/10.1109/FOCS.2016.69

[9] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact. *Empirical Softw. Engg.* 10, 4 (Oct. 2005), 405–435. https://doi.org/10.1007/s10664-005-3861-2

[10] R Dum, P Zoller, and H Ritsch. 1992. Monte Carlo simulation of the atomic master equation for spontaneous emission. *Physical Review A* 45, 7 (1992), 4879.

[11] 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

[12] Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic Symbolic Execution. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (Minneapolis, MN, USA) *(ISSTA 2012)*. Association for Computing Machinery, New York, NY, USA, 166–176. https://doi.org/10.1145/2338965.2336773

[13] Mitchell Gerrard, Mateus Borges, Matthew B. Dwyer, and Antonio Filieri. 2022. Conditional Quantitative Program Analysis. *IEEE Transactions on Software Engineering* 48, 4 (2022), 1212–1227. https://doi.org/10.1109/TSE.2020.3016778

[14] Carla P Gomes, Ashish Sabharwal, and Bart Selman. 2021. Model counting. In *Handbook of satisfiability*. IOS press, 993–1014.

[15] I. J. Good. 1953. The Population Frequencies of Species and the Estimation of Population Parameters. *Biometrika* 40, 3/4 (1953), 237–264. http://www.jstor.org/stable/2333344

[16] William Landi. 1992. Undecidability of static analysis. *LOPLAS* 1 (1992), 323–337.

[17] Seongmin Lee. 2022. *Statistical program dependence approximation*. Ph. D. Dissertation. Korea Advanced Institute of Science and Technology (KAIST), Daejeon.

[18] S. Lee, D. Binkley, R. Feldt, N. Gold, and S. Yoo. 2019. MOAD: Modeling Observation-Based Approximate Dependency. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 12–22. https://doi.org/10.1109/SCAM.2019.00011

[19] Seongmin Lee, David Binkley, Robert Feldt, Nicolas Gold, and Shin Yoo. 2021. Observation-based approximate dependency modeling and its use for program slicing. *Journal of Systems and Software* 179 (2021), 110988. https://doi.org/10.1016/j.jss.2021.110988

[20] Danushka Liyanage, Marcel Böhme, Chakkrit Tantithamthavorn, and Stephan Lipp. 2023. Reachable Coverage: Estimating Saturation in Fuzzing. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. 1–13.

[21] Kasper Luckow, Corina S Păsăreanu, and Willem Visser. 2018. Monte Carlo tree search for finding costly paths in programs. In *International Conference on Software Engineering and Formal Methods*. Springer, 123–138.

[22] Metodi Mazhdrakov, Dobriyan Benov, and Nikolai Valkanov. 2018. *The Monte Carlo method: engineering applications*. ACMO Academic Press.

[23] Nikhil Parasaram, Earl T. Barr, Sergey Mechtaev, and Marcel Böhme. 2023. Precise Data-Driven Approximation for Program Analysis via Fuzzing. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE 2023)*. Association for Computing Machinery, 1–12.

[24] 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* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 179–180. https://doi.org/10.1145/1858996.1859035

[25] Seemanta Saha. 2022. *PReach: A probabilistic reachability analyzer to identify hard to reach program statements.* https://doi.org/10.5281/zenodo.5915206

[26] Seemanta Saha, Mara Downing, Tegan Brennan, and Tevfik Bultan. 2022. PReach: A Heuristic for Probabilistic Reachability to Identify Hard to Reach Statements. In *International Conference on Software Engineering (ICSE)*.

[27] Mathematics UC Davis. [n. d.]. *Latte integrale.* http://www.math.ucdavis.edu/~latte

[28] 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.

[29] Edwin B Wilson. 1927. Probable inference, the law of succession, and statistical inference. *J. Amer. Statist. Assoc.* 22, 158 (1927), 209–212.

[30] Lei Zhao, Yue Duan, Heng Yin, and J. Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. *Proceedings 2019 Network and Distributed System Security Symposium* (2019).

Received 2023-02-02; accepted 2023-07-27