Symbolic Execution for Randomized Programs

ANONYMOUS AUTHOR(S)

We propose a symbolic execution method for programs that can draw random samples. Crucially, and in contrast to existing work, our method can handle randomized programs with unknown inputs and can prove probabilistic properties that universally quantify over all possible inputs. Our technique augments standard symbolic execution with a new class of *probabilistic symbolic variables*, which represent the results of random draws, and data structures for tracking the distribution of random samples. Our approach computes symbolic expressions representing the probability of taking individual paths. We implement our method on top of the KLEE symbolic execution engine, and use it to prove properties about probabilities and expected values for a range of challenging case studies written in C++, including Freivalds' matrix multiplication verification algorithm, randomized quicksort, and a randomized property-testing algorithm for monotonicity.

Additional Key Words and Phrases: Probabilistic programs, symbolic execution

1 INTRODUCTION

Symbolic execution (SE) [King 1976] is a highly successful method to automatically find bugs in programs. In a nutshell, SE iteratively explores the space of possible program paths, while treating program states as *symbolic functions* of program inputs (symbolic parameters). Whenever SE discovers a path to an error state, SE checks to see if there is a setting of the symbolic parameters that can execute along this path, thereby triggering an error. This basic bug-finding strategy has proved to be a powerful technique, supporting some of the most effective tools for analyzing large codebases [Bessey et al. 2010].

Aiming to expand the reach of SE, researchers have proposed many domain-specific extensions of SE (e.g., Borges et al. [2014]; Farina et al. [2019]; Filieri et al. [2013]; Geldenhuys et al. [2012]; Kang et al. [2021]; Kiezun et al. [2013]; Sasnauskas et al. [2011, 2010]). In this vein, we consider how to perform SE for *randomized* programs, which can draw random samples from built-in distributions. These programs play a critical role in many leading applications today, from machine learning to security and privacy. Randomized programs, like all programs, are susceptible to bugs [Joshi et al. 2019]. Correctness properties are difficult to formally verify; existing methods typically require substantial manual effort and target highly specific properties. Moreover, these programs are highly difficult to test: the desired behavior is not deterministic, and it is hard to tell if a program is producing the wrong distribution of outputs just by observing executions.

Challenges and prior work. Accordingly, randomized programs are an attractive target for general-purpose, automated verification methods, like SE. However, there are numerous challenges in developing an SE procedure for probabilistic programs:

- **Quantitative properties.** In standard SE, the goal is to identify whether a bad program state (e.g., an assertion failure) is reachable or not—if so, the program has a bug. In randomized programs, we are often more interested in whether a bad state is reached *too often*, or whether a good state is reached *often enough*. Accordingly, a useful SE procedure must be able to analyze the quantitative probability of reaching certain program states.
- **Computing branch probabilities.** A concrete execution of a non-probabilistic program takes a single branch at each branching instruction, since the branch condition is either true or false in any program state. When reasoning about probabilistic behavior, it is more useful to model the program as transforming a distribution over program states. Then, a branch condition has some probability of being true, and some probability of being false, in

2022. 2475-1421/2022/1-ART1 $15.00
https://doi.org/

every probabilistic program state. A SE procedure for probabilistic programs should be able to compute probabilities of branches; by combining these probabilities along a path, the SE procedure can then reason about the probabilities of paths.

- **Handling program inputs.** Like standard programs, randomized programs often have input parameters. These unknown values are qualitatively different from the unknown results of random sampling commands: there is often no sensible probability one can assign to different settings of the program inputs, and the target correctness property usually universally quantifies over all possible inputs.
  Prior work has considered SE for probabilistic programs [Geldenhuys et al. 2012; Sampson et al. 2014], but unlike traditional SE, these methods do not treat the program inputs as unknown values. Instead, existing methods consider probabilistic programs without inputs [Sampson et al. 2014], or assume that the inputs are probabilistically drawn from a known distribution [Geldenhuys et al. 2012]. This simplification allows SE to compute probabilities exactly using techniques like model counting, or approximately using statistical sampling. However, it significantly limits the kinds of programs that can be analyzed, and the kinds of properties that can be proved.

Our work: symbolic execution for randomized programs. We propose a symbolic execution method for randomized programs, where programs may have unknown inputs. These programs can be thought of as producing a family of output distributions, one for each concrete input, and inputs can range over a large, conceptually infinite set. We consider two kinds of properties:

- **Probability bounds.** For all inputs, the *probability of reaching a program state* is at most/at least/exactly equal to some quantity, which may depend on the inputs. This class of properties is the probabilistic analogue of the reachability properties typically considered by SE.
- **Expectation bounds.** For all inputs, the *expected value of a program expression* in the output is at most/at least/exactly equal to some quantity, which may depend on the inputs. This class of properties is more general than probability bounds, and is useful for reasoning about quantitative properties like expected resource usage.

There are two key technical ingredients in our approach:

- **Distinguish between regular and probabilistic symbolic variables.** We model probabilistic sampling statements by introducing a new kind of symbolic variable, referred as *probabilistic symbolic variable*. While regular symbolic variables represent unknown inputs, probabilistic symbolic variables represent the result from sampling from a known distribution. Prior work only supports one or the other kind of symbolic variable; our method supports both simultaneously.
- **Compute symbolic branch probabilities.** In the presence of program inputs, the probability of taking a branch may depend on unknown values. Accordingly, it is not possible to use approaches like model counting to compute the branch probabilities, since the probabilities are not constant numbers. Instead, we compute the branch probability expressions symbolically.

Contributions and outline. After illustrating our method on a motivating example (§ 2), we present our main contributions:

- A symbolic execution method for probabilistic programs with unknown input parameters, along with a formal proof of soundness for the extensions to the probabilistic case (§ 3).
- A broad collection of case studies drawn from the randomized algorithms literature (§ 4). All case studies have unknown input parameters, and none of the examples can be handled using

existing automated methods. Some example properties include bounding the soundness probability of Freivalds' algorithm [Freivalds 1977], a randomized algorithm to check matrix multiplication; the expected number of comparisons for randomized quicksort; and the correctness probability of a randomized property-testing algorithm for checking monotonicity [Goldreich 2017].

- An implementation of our approach called Plinko, building on the KLEE execution engine [Cadar et al. 2008], and an evaluation on our case studies (§ 5). By building on KLEE, Plinko is able to perform symbolic execution on real implementations of probabilistic programs with natural code written in a wide variety of mainstream languages (e.g., anything targeting LLVM), while faithfully models realistic program states (e.g., with overflowing arithmetic, arrays, etc.).

We discuss related work in § 6, and future directions in § 7. *An extended version of the paper that includes all the detailed proofs (in the appendix) is provided as supplemental material.*

2 OVERVIEW

The Monty Hall problem [Selvin 1975] is a classic probability puzzle based on the American television show, *Let's Make a Deal*, which showcases how subtle probabilistic reasoning can be. The problem itself is simple:

> You are a contestant on a game show and behind one of three doors there is a car and behind the others, goats. You pick a door and the host, who knows what is behind each of the doors, opens a different door, which has a goat. The host then offers you the choice to switch to the remaining door. Should you?

While it may seem unintuitive, regardless of the contestant's original door choice, the contestant who always switches doors will win the car $\frac{2}{3}$ of the time, as opposed to a $\frac{1}{3}$ chance if the contestant sticks with their original choice.

We can represent this problem as a probabilistic program, as shown in Figure 1, where `choice` $\in [1, 3]$ is the door which is originally chosen by the contestant and `door_switch` is `true` if the contestant wants to switch doors when asked, and `false` otherwise. If `door_switch` = `true`, regardless of the value of `choice`, `monty_hall` should return `true` (i.e., the car is won) $\frac{2}{3}$ of the time. In general, the problem we aim to solve is: given a probabilistic program with discrete sampling statements, and a target probability bound, how do we verify that the program satisfies the bound? For this particular program, the property is not so hard to verify with existing methods—the input space is tiny, so it is feasible to try all possible inputs and verify the probability bound on each

```
1  int monty_hall(int choice, bool door_switch) {
2    int car_door = uniform_int(1,3);
3    int host_door;
4    if (choice == car_door)
5      return !door_switch;
6    if (choice != 1 && car_door != 1)
7      host_door = 1;
8    else if (choice != 2 && car_door != 2)
9      host_door = 2;
10   else
11     host_door = 3;
12   if (door_switch)
13     if (host_door == 1)
14       if (choice == 2)
15         choice = 3;
16       else
17         choice = 2;
18     else if (host_door == 2)
19       if (choice == 1)
20         choice = 3;
21       else
22         choice = 1;
23     else
24       if (choice == 1)
25         choice = 2;
26       else
27         choice = 1;
28   return choice == car_door;
29 }
```

Fig. 1. C code for the Monty Hall problem.

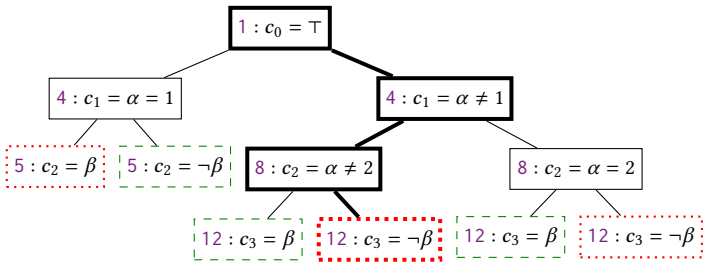


Fig. 2. Execution Tree for the Monty Hall Problem if the car is behind door 1.

output distribution. However, we use this example to illustrate our symbolic execution technique, which will scale to the more complex examples we will see in § 4.

In symbolic execution, program inputs are replaced by *symbolic* variables which can take on any value. The program is then "run" on these symbolic variables and when a branch is reached execution proceeds along each of these two branches and the constraint is recorded in that path's, *path condition*, which is represented by φ. This yields a *symbolic execution tree* that represents the possible paths through the program. Suppose we were to run traditional symbolic execution on the program in Figure 1 where the car was randomly chosen to be behind door 1. The two inputs, `choice` and `door_switch`, would become the symbolic variables α and β, respectively. The execution tree is shown in Figure 2. Symmetric trees can be made for the cases when the car is behind door 2 and door 3. For each node in the tree the line number of the branch statement and the branch condition is given. Leaves which are surrounded by a dashed (`_ _`), green line represent the contestant winning the car (the function returning `true`), and those leaves which are surrounded by a dotted (….), red line represent the contestant losing the contest (the function returning `false`). Each path condition, φ, can then be written as a conjunction of the c_is along a path from the root of the tree to a leaf.

For example, consider the **bolded** path through the tree. The execution begins on line 1 with an empty path condition, $c_0 = \top$. Execution then reaches the first **if** condition on line 4 and takes false branch which jumps to line 6; however, the guard, `choice != 1 && car_door != 1` is always `false` as `car_door == 1`. We then skip this branch and instead branch again on line 8, and take the true branch to get to the third node in the bolded path. Lastly, execution jumps to line 12 where the contestant's choice to switch doors is checked. This branch actually determines whether the car is won along this path as we already have determined that the contestant did not choose the door which hides the car on line 4. If the contestant doesn't switch, they will lose, and since at this stage the host has revealed a door which has a goat behind it, the remaining door *must* have the car. Therefore, switching will win the contestant the car and so this path ends in a loss. The entire path condition, φ, can then be written as $\varphi = \bigwedge_i c_i = \alpha \neq 1 \wedge \alpha \neq 2 \wedge \neg\beta$.

While this tree can tell us that it is *possible* to win (or lose) the car with any choice of initial door and decision to switch, it does not tell us how *often* a contestant will win for any initial door and decision to switch. In theory, if we knew how likely it was to take one branch over another we could extend this reasoning to determine how often we would hit a winning leaf over a losing leaf. **Our core idea is to represent probabilistic sampling statements as a new class of symbolic variables.** For the sake of presentation we refer to these as *probabilistic* symbolic variables as opposed to, what we call, *universal* symbolic variables which range over all possible values. So, instead of asking for a random value, we replace the result of the sample with a

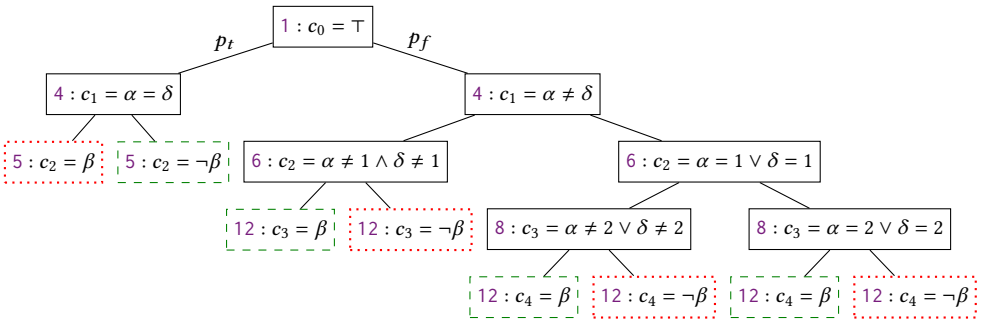


Fig. 3. Execution tree for the Monty Hall Problem with probabilistic symbolic variables.

probabilistic symbolic variable and record the distribution from which the sample is from. Using this information, we can then compute the probabilities of taking branches.

Now let us return to the Monty Hall problem from Figure 1. Let `car_door` be represented by the probabilistic symbolic variable, δ and let `choice` and `door_switch` be α and β, respectively, as before. Instead of branching solely on α and β, we additionally branch on δ. The execution tree is presented in Figure 3. Note that branches are omitted from the tree if the direction to traverse can be inferred by the current path condition.

Since probabilistic symbolic variables originate from a distribution, we can determine the *probability* of taking a certain branch by simply counting how many values from the distribution satisfy the guard condition. For example, to figure out the probability of taking the `true` branch of the `if` condition on line 4 (denoted by p_t in Figure 3), it suffices to count how many settings of δ satisfy $\alpha = \delta$ and divide by the total number of possible assignments to δ (i.e., 3). However, since α is a symbolic variable we can only obtain a probability expression which is in terms of universal symbolic variables. Let $[\cdot]$ denote Iverson brackets, where $[Q] = 1$ if formula Q is true, and 0 otherwise. Then, $p_t = ([\alpha = 1] + [\alpha = 2] + [\alpha = 3])/3 = \frac{1}{3}$ as $\delta \in [1, 2, 3]$ and each setting is equally likely. Similarly, $p_f = ([\alpha \neq 1] + [\alpha \neq 2] + [\alpha \neq 3])/3 = \frac{2}{3}$. We stress that these probabilities are *symbolic expressions*: they may depend on the universal symbolic variables. This feature is one of the key advantages of our technique over existing methods for probabilistic symbolic execution—it makes reasoning about the probabilities much more complex, but it also enables proving properties for programs with unknown inputs. (We defer further comparison with related work to § 6.)

With this *probabilistic* execution tree in hand, let's return to the original question we wanted to answer: if the contestant switches doors, does their chances of winning the car exceed $\frac{1}{3}$? Note that each path in the tree has a probability associated with it. If we want to know the probability of winning if the contestant switches versus not, we can look at solely those paths in the tree which lead to a win. To then figure out the probability of winning the car if the contestant switches we then can remove those paths where $\neg\beta$ is true, and then sum up the probabilities of the remaining paths. This results in an expression in terms of α and constants, which we can then evaluate where $\alpha = 1$, $\alpha = 2$, and $\alpha = 3$ and compare the probabilities. In the end, we get that regardless the setting of α (i.e., the program variable, `choice`) the probability of winning the car if the contestant switches doors is exactly $\frac{2}{3}$. We formalize this intuition in Section 3.3 by expressing this query in first order logic, which can then be dispatched to automated solvers.

3 PROBABILISTIC SYMBOLIC EXECUTION ALGORITHM

We present the formalism of our approach on **pWhile**, a core imperative probabilistic programming language, as a model for more general probabilistic languages. The statements of **pWhile** are described by the following grammar:

$$S := \mathsf{x} \leftarrow e \mid \mathsf{x} \sim d \mid S_1; S_2 \mid \textbf{if } c \textbf{ then goto } T \mid \textbf{halt}$$

Intuitively, the assignment statement $\mathsf{x} \leftarrow e$ assigns the result of evaluating the expression e to the program variable x, while the sampling statement $x \sim d$ draws a random sample from a primitive distribution d and assigns the result to the program variable x. Here, d is a *discrete* distribution expression which denotes which distribution the sample should be drawn from. We interpret distributions as functions from values to the range $[0, 1]$, which denotes the probability of the value occurring in the distribution. For example, $\mathsf{UniformInt}(1, 6)$ is a uniform distribution which selects at random a value between 1 and 6 (inclusive).

Control flow is implemented by $S_1; S_2$, which sequences two statements S_1 and S_2, and conditional branching **if** c **then goto** T, which jumps execution to statement T if the guard c holds, otherwise falling through to the next instruction. We assume that high-level constructs like loops and regular conditionals are compiled down to conditional branches. Finally, **halt** marks the end of execution.

Our probabilistic symbolic execution algorithm will aim to answer the following question: What is the maximum (or minimum) probability that a program, Prog, terminates in a state where a predicate ψ holds? Formally, we are interested in computing

$$\max_{\vec{x}} \{\Pr_{\mu}[\psi] \mid \mu = \mathsf{Prog}(\vec{x}, \psi)\}$$

where μ is the distribution on outputs obtained by running Prog on inputs $\vec{x}$. Since this quantity is difficult to compute in general, we will also be interested in computing bounds on this quantity:

$$\max_{\vec{x}} \{\Pr_{\mu}[\psi] \mid \mu = \mathsf{Prog}(\vec{x}, \psi)\} \bowtie f(\vec{x})$$

where $\bowtie \in \{\leq, \geq, =, \ldots\}$ is a comparison operator and f is some given function of the program inputs.

We begin with an overview of standard symbolic execution for non-probabilistic programs (§ 3.1). Then, we present our symbolic execution approach for probabilistic programs. Our approach has two steps: first, we augment standard symbolic execution to track probabilistic information as paths are explored (§ 3.2). Then, we convert program paths into a logical query encoding a probabilistic property (§ 3.3). Finally, we formalize the semantics of our approach and prove its soundness (§ 3.4).

3.1 Symbolic Execution

Symbolic execution [King 1976] is a program analysis technique that iteratively explores the set of paths through a given program. Since program paths may depend on input variables, which are not known, symbolic execution treats each input as a *symbolic* variable. All program operations are then redefined to manipulate these symbolic variables. We present a high-level description of symbolic execution in Algorithm 1. The reader should ignore portions in shaded boxes for now; these are our extensions to handle probabilistic programs, which we will discuss in the next section.

The symbolic execution algorithm tracks information about program states using the following key data-structures: the next instruction to be executed, I_c, a conjunctive formula, φ, consisting of the symbolic branch constraints that are true for a particular path, called a *path condition*, and a mapping from program variables to symbolic expressions over constants and symbolic variables, σ, called an *expression map*. To begin, Algorithm 1 initializes the execution stack, E_s, with an empty list, and the expression map, σ with fresh symbolic variables for each *program input* in the input vector,

Algorithm 1 Probabilistic Symbolic Execution Algorithm

```
1:  procedure SymbEx(Prog : Program, x⃗ : Program Inputs, ψ : Predicate)
2:      E_s ← [], σ ← BindToSymbolic(x⃗), Enc_ψ ← 0.0, Enc_a ← 0.0        ▷ Initialization
3:      I_0 ← getStartInstruction(Prog)
4:      S_0 ← (I_0, ⊤, ∅, ∅, 1.0)                                        ▷ Empty Initial State
5:      E_s.Append(S_0)                                                  ▷ Start with S_0 in Execution Stack
6:      while E_s ≠ ∅ do
7:          (I_c, φ, σ, P, p) ← E_s.Remove( )
8:          if UNSAT(φ) then
9:              continue
10:         switch instType(I_c) do
11:             case x ← e                                               ▷ Assignment Instruction
12:                 I_1 ← getNextInstruction(I_c)
13:                 σ[x] ← σ[e]
14:                 S_1 ← (I_1, φ, σ, P, p)
15:                 E_s.Append(S_1)
16:             case x ~ d                                               ▷ Sampling Instruction
17:                 I_1 ← getNextInstruction(I_c)
18:                 σ_1, P_1 ← PSESample (x, d, σ, P)
19:                 S_1 ← (I_c, φ, σ_1, P_1, p)
20:                 E_s.Append(S_1)
21:             end case
22:             case if c then goto T                                    ▷ Branch Instruction
23:                 c_sym ← σ[c]
24:                 p_t, p_f ← PSEBranch (c_sym, φ, P)
25:                 I_1 ← getInstruction(T)
26:                 I_2 ← getNextInstruction(I_c)
27:                 S_t ← (I_1, φ ∧ c_sym, σ, P, p · p_t)                ▷ Multiply with current path probability
28:                 S_f ← (I_2, φ ∧ ¬c_sym, σ, P, p · p_f)
29:                 E_s.Append(S_f)
30:                 E_s.Append(S_t)
31:             case halt                                                ▷ Terminate Instruction
32:                 ψ_sym ← σ[ψ]
33:                 if SAT(ψ_sym) then
34:                     Enc_ψ ← Enc_ψ + p
35:                 end if
36:                 Enc_a ← AddEncoding(Enc_a, p, σ)
37:     r ← Solve (Enc_ψ, Enc_a)
38:     return r
```

$\vec{x}$ by calling BindToSymbolic as shown in line 2. I_0 is then initialized with the first instruction of the program at line 3 and the initial state tuple S_0 is created on line 4 and appended to E_s.

The core of Algorithm 1 is presented on lines 6 to 36 as a loop which iterates through all reachable states until E_s is exhausted. For each iteration we pop a state from E_s (line 7) and, if the selected

path condition φ is feasible, the target instruction is analyzed based on the instruction's form: line 11 for assignment statements and line 22 for branch statements.

For an assignment statement, $x \leftarrow e$, where e is a program expression, we first convert e into a *symbolic* expression using the expression map, σ. We use the notation $\sigma[e]$ to denote the symbolic expression e_{sym} constructed by replacing each program variable in e with its corresponding symbolic expression in σ. For a branch statement, **if** e **then goto** T, the symbolic execution algorithm *forks* by constructing the corresponding path conditions for both branches, appending the symbolic branch guard c_{sym} in the positive or negative form (for the true and false directions, respectively). Finally, symbolic execution along a path terminates at a **halt** instruction, and the terminating states may be appended to an encoding for subsequent analysis.

3.2 Probabilistic Symbolic Execution

Our key idea is to enable symbolic execution over probabilistic constructs using a new category of symbolic variables, which we refer to as *probabilistic symbolic variables*. Accordingly, our algorithm distinguishes between two categories of symbolic variables:

- *Universal symbolic variables*: These are identical to those in traditional symbolic execution, and are used to model arbitrary program inputs.
- *Probabilistic symbolic variables*: Probabilistic symbolic variables model a known distribution over a set of values. In our algorithm, a new probabilistic symbolic variable is created for each sampling statement to denote the sampling operation.

Symbolic execution becomes quite challenging when both universal and probabilistic variables are present, as now program branches are taken with some quantitative probability. To track information about probabilistic states, we add the following data-structures:

- *Distribution map* (P). This map from probabilistic symbolic variables to distribution expressions tracks the distribution from which a probabilistic symbolic variable was originally sampled from. This is analogous to the symbolic variable map, σ, except for mapping probabilistic symbolic variables to distributions instead from program variables to symbolic expressions.
- *Path probability* (p). For each path, we adjoin a path probability expression, p, which may be parameterized by universal symbolic variables. Now each path has an associated path condition and path probability.

With this new infrastructure, we can now define how symbolic execution proceeds when execution reaches a sampling statement, and demonstrate how we compute the probability of taking either side of a branch statement.

Algorithm 2 PSE Sampling Algorithm

```
1: function PSESample(x, d, σ, P)
2:     δ ← Generate a fresh probabilistic symbolic variable
3:     σ[x] = δ
4:     P[δ] = d
5:     return (σ, P)
```

Sampling (Lines 16 to 21). On encountering a sampling statement, $x \sim d$, at line 16 of Algorithm 1, we call PSESample to get the updated distribution map, P, and expression map, σ. The function PSESample (Algorithm 2) takes four inputs: a program variable, x, a distribution expression, d, which determines the distribution the value of x is sampled from, the current expression map,

σ, and the current distribution map, P, and returns updated expression and distribution maps. To model the random value returned from the distribution expression d, a fresh probabilistic symbolic variable, δ, is created and x is mapped to δ under σ. Additionally, the distribution map is updated to reflect that δ is "drawn" from d.

Algorithm 3 PSE Branch Algorithm

1: **function** PSEBranch(c_{sym}, φ, P)
2: $(\delta_1, \ldots, \delta_n) \leftarrow \text{dom}(P)$
3: $(d_1, \ldots, d_n) \leftarrow (P[\delta_1], \ldots, P[\delta_n])$ ▷ Retrieve all distribution expressions
4: $$p_c \leftarrow \frac{\sum_{(v_1,\ldots,v_n)\in \text{dom}(d_1)\times\cdots\times\text{dom}(d_n)} [(c_{sym} \wedge \varphi)\{\delta_1 \mapsto v_1, \ldots, \delta_n \mapsto v_n\}]}{\sum_{(v_1,\ldots,v_n)\in \text{dom}(d_1)\times\cdots\times\text{dom}(d_n)} [\varphi\{\delta_1 \mapsto v_1, \ldots, \delta_n \mapsto v_n\}]}$$
5: **return** $(p_c, 1 - p_c)$

Branches (Lines 22 to 30). Since probabilistic programs can branch on random values, each branch has a probability of being taken. Note that with the addition of probabilistic symbolic variables we can now either branch on universal or probabilistic symbolic variables (or both).

Upon reaching a branch statement, similar to traditional symbolic execution, the symbolic states corresponding to the true and false branches are added to E_s. Then, PSEBranch (Algorithm 3) computes probability *expressions* for the true and false branches. PSEBranch takes as input a symbolic expression, c_{sym}, which is equivalent to the guard expression c, the current path condition, φ, and the current distribution map, P, and returns two probability expressions: p_c, which is denotes the probability of taking the true branch, and $1 - p_c$, which denotes taking the false branch. For simplicity, we present this subroutine in the simplified setting where all sampling instructions are from the uniform distribution over a finite set; handling general weighted distributions is not much more complicated.

To gain intuition on the computation of path probabilities, for now just consider the special case where the guard condition only involves universal symbolic variables. Let $c_{sym} = \sigma[c]$ be the equivalent symbolic expression for the guard c and assume that c_{sym} does not mention any probabilistic symbolic variables. In this case, the branch that is taken is entirely determined by the universal symbolic variables. Therefore, for a fixed setting of the universal symbolic variables, one side of the branch must have a probability of 1, and the other side, 0. Put another way, either c_{sym} can be satisfiable or $\neg c_{sym}$, but never both. We use Iverson brackets to formalize this idea; the probability of taking the "true" branch is $[c_{sym}]$, and the probability of taking the "false" branch is $[\neg c_{sym}]$.

For probabilistic branches (i.e., guards which branch on probabilistic symbolic variables), computing the branch probability is more involved. We give an intuitive justification here, and defer the formal proof of correctness to § 3.4. When symbolic execution is at the start of a conditional branch statement, the path condition φ records the necessary constraints on the universal and probabilistic symbolic variables which must hold in order to reach this conditional branch. Then, we can view the probability of taking either branch as a *conditional probability*: the probability that c_{sym} holds assuming that φ is satisfied. In more formal notation, we aim to compute $\Pr[c_{sym} \mid \varphi] = \Pr[c_{sym} \wedge \varphi]/\Pr[\varphi]$.

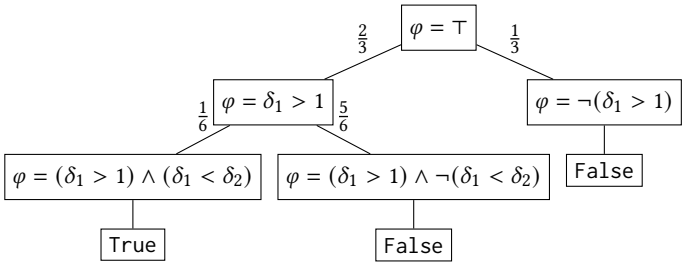
Algorithm 3 builds an expression which computes this conditional probability. Note that each probabilistic symbolic variable, δ, is mapped to exactly one distribution, d, and therefore, $\delta \in$

dom (d). So, assuming there are n probabilistic symbolic variables, $\delta_1, \ldots, \delta_n$, and so n distributions, $d_1, \ldots, d_n$, the set of all possible values for $\delta_1, \ldots, \delta_n$ is $\mathcal{D} = \text{dom}\,(d_1) \times \cdots \times \text{dom}\,(d_n)$. Using our simplifying assumption that all probabilistic symbolic variables are drawn from a uniform distribution, each assignment $v_1, \ldots, v_n$ to $\delta_1, \ldots, \delta_n$ has equal probability. Thus, instead of computing the conditional probability as a ratio of probabilities, we can compute the ratio of the number of assignments from $\mathcal{D}$ which satisfy $c_{sym} \wedge \varphi$ and φ, as shown on line 4 of Algorithm 3. Note that p_c is not necessarily a value, but rather a symbolic expression containing constants and universal symbolic variables. Finally, we use the fact that the sum of the conditional probabilities of the branch outcomes is 1, which allows us to avoid computing the probability of taking the false branch directly.

Now that we know the probability of taking a branch, we can compute the probability of traversing the entire path, or equivalently, the probability of $\varphi \wedge c_{sym}$ (or $\varphi \wedge \neg c_{sym}$) being satisfied. Note that p_t is the *conditional* probability of taking the true branch and p_f is the conditional probability of taking the false branch on line 24, or $p_t = \Pr[c_{sym} \mid \varphi]$ and $p_f = \Pr[\neg c_{sym} \mid \varphi]$. Since p is the probability of φ, to the probability of $\varphi \wedge c_{sym}$, we can simply multiply $p \cdot p_t$ according to the standard rule $\Pr[A \wedge B] = \Pr[A \mid B] \Pr[B]$ where A and B are any two events. The same can be done for $\varphi \wedge \neg c_{sym}$, as shown on lines 27 and 28.

```
1: x ~ UniformInt(1, 3)
2: y ~ UniformInt(1, 3)
3: if x > 1 then
4:     if x < y then
5:         return True
6: else
7:     return False
```

(a) Program with random sampling.



(b) Symbolic execution tree.

Fig. 4. A sample, randomized program and its associated symbolic execution tree annotated with probabilities.

Example. Consider the code in Figure 4a and suppose we wish to calculate the probability of the program returning `True`. Following Algorithm 2, we generate fresh probabilistic symbolic variables for x and y, δ_1 and δ_2, respectively. We also store the distributions which δ_1 and δ_2 are samples from, namely the discrete uniform distribution $\mathcal{U}\{1,3\}$. In our notation, we say that $\sigma = \{\mathsf{x} \mapsto \delta_1, \mathsf{y} \mapsto \delta_2\}$ and $P = \{\delta_1, \delta_2 \mapsto \mathcal{U}\{1,3\}\}$. Now following Algorithm 3 to process line 3 of Figure 4a, note that $\mathcal{D} = \{1,2,3\} \times \{1,2,3\}$ and

$$p_c = \Pr[\delta_1 < \delta_2 \mid \delta_1 > 1] = \frac{\Pr[(\delta_1 < \delta_2) \wedge (\delta_1 > 1)]}{\Pr[\delta_1 > 1]}$$
$$= \frac{\sum_{(v_1,v_2)\in\mathcal{D}} [(\delta_1 < \delta_2) \wedge (\delta_1 > 1)\{\delta_1 \mapsto v_1, \delta_2 \mapsto v_2\}]}{\sum_{(v_1,v_2)\in\mathcal{D}} [(\delta_1 > 1)\{\delta_1 \mapsto v_1\}]} = \frac{1}{6}$$

This probability means that the probability of taking the true branch of the inner **if** condition is only $\frac{1}{6}$, which makes sense as x is restricted to be either 2 or 3, but y can be either 1,2, or 3; however, only one combination of x and y will satisfy x < y, namely x = 2 and y = 3.

3.3 Query Generation

Recall that our original goal is to find the maximum (or minimum) probability that a program, Prog, terminates in a state where a predicate ψ holds. We frame this question as a logical query using the symbolic state found in the execution tree. In general, our queries are of the following form:

$$\forall \alpha_1, \ldots, \alpha_n \,.\, Enc_\psi \bowtie f(\alpha_1, \ldots, \alpha_n) \tag{1}$$

where $\alpha_1, \ldots, \alpha_n$ are *all* the universal symbolic variables found in Prog, Enc_ψ is an expression representing the probability that ψ holds in Prog, $\bowtie$ is a binary relation (e.g., $>, <, \geq, \leq$), and f is some function of the universal symbolic variables denoting the desired bound to check. There are three main components to our queries: 1) a universal quantification over the universal symbolic variables, 2) Enc_ψ, or the probability that ψ holds in Prog, and 3) a desired upper/lower bound expression, potentially parameterized by universal symbolic variables, e

(1) If a program has n universal symbolic variables, $\alpha_1, \ldots, \alpha_n$, we begin the query with a universal quantification, $\forall \alpha_1, \ldots, \alpha_n$, in order to reason over any setting of the non-probabilistic program variables.
(2) We construct Enc_ψ on lines 32 to 36 of Algorithm 1. Once a path reaches a **halt** statement, ψ is converted into a symbolic expression using the expression map, σ, called ψ_{sym}. If ψ_{sym} is satisfiable, then we add p to the current Enc_ψ as this path's probability should be included in our query. Note that during some settings of $\alpha_1, \ldots, \alpha_n$, the path represented by φ might not be reachable, and so we should exclude that probability from the sum. However, since we use φ in the construction of p, if φ is unsatisfiable, then p will be 0 as each summand in the numerator of p_c, as defined on line 4 of Algorithm 3 will be 0. For example, recall the Monty Hall program in Figure 1 and consider the left-most two paths in Figure 3, namely $\varphi_1 := \alpha = \delta \wedge \beta$ and $\varphi_2 := \alpha = \delta \wedge \neg\beta$. Note that $p_1 = [\alpha = 1 \wedge \beta] + [\alpha = 2 \wedge \beta] + [\alpha = 3 \wedge \beta]/3$ and $p_2 = [\alpha = 1 \wedge \neg\beta] + [\alpha = 2 \wedge \neg\beta] + [\alpha = 3 \wedge \neg\beta]/3$. If β is true, then $p_2 = 0$ regardless of the setting of α, as φ_2 is unsatisfiable. Once we have exhausted all reachable states in E_s, Enc_ψ represents the probability that ψ holds for all reachable paths in Prog.
(3) Let f be the lower/upper bound function of universal symbolic variables which we want to prove Prog does not violate. Another way of interpreting f is a function which computes the *desired* probability of ψ occurring in Prog (e.g., a maximum acceptable error rate) given a setting of the universal symbolic variables. We should note, however, that symbolic execution is not complete, meaning not all possible paths necessarily will be reached. Therefore, we can only find violations on upper bounds, but not lower bounds, since it is possible that some missing paths might violate the lower bound, but if we find a violation on an upper bound, more paths will only make that violation more egregious.

Once probabilistic symbolic execution terminates, Solve is called on line 37 which takes in Enc_ψ, constructs formula 1, and calls an automated decision procedure, such as an SMT solver, to answer the query. The meaning of this query depends on the termination condition of symbolic execution. If all reachable paths are explored, then Enc_ψ represents the true probability that ψ holds in the output distribution, parameterized by the unknown input variables. However, in most realistic settings, symbolic execution will not be able to explore all paths due to resource and time limitations. In this case, Enc_ψ may not be equal to the true probability of ψ; however, it is always a sound *lower bound* of this probability. This information is enough for our approach to refute upper

bounds on probability: if we want to verify that the probability of ψ is at most $f(\alpha_1, \ldots, \alpha_n)$ for all settings of the input variables, and symbolic execution produces a probability mass Enc_ψ that exceeds $f(\alpha_1, \ldots, f_n)$ for some setting of the input variables, then the original upper bound cannot hold.

Example. In § 2 we introduced the Monty Hall problem, and we wanted to prove that if a contestant always switched doors their probability of winning the car is $\frac{2}{3}$. With can frame this question as a query of the form defined in Equation (1). If α is the universal symbolic variable corresponding to `choice`, and β is the universal symbolic variable corresponding to `door_switch`, $\psi := \beta \wedge \mathsf{win}$ where win means the car is won (determined by the return value of `monty_hall`), $\bowtie := =$, and $f := \frac{2}{3}$, then our full query is: $\forall \alpha, \beta \,.\, Enc_\psi = \frac{2}{3}$. Algorithm 1 then constructs the tree found in Figure 3 and constructs the probability expression for each path. If φ_i is path condition for the path ending in the i^{th} left-most leaf of Figure 3, and p_i is the probability expression for φ_i, as created by Algorithm 1, then $Enc_\psi = p_3 + p_5 + p_7$. Solve then takes the query $\forall \alpha, \beta \,.\, p_3 + p_5 + p_7 = \frac{2}{3}$ and uses an external automated decision procedure to prove for each possible setting of α and β that $p_3 + p_5 + p_7 = \frac{2}{3}$.

Algorithm 4 AddEncoding for computing $\mathbb{E}[\mathsf{v}]$

```
1: function AddEncoding(Enc_a, p, σ)
2:     return Enc_a + p · σ[v]
```

3.3.1 Expected Value Queries. In addition to normal probability bound queries, our technique can also be used to compute the expected value of any variable with only a slight modification. Suppose that we want to compute the expected value of a program variable v, denoted $\mathbb{E}[\mathsf{v}]$. Since each path has a probability, p_i, and an expression map, σ_i, we can construct a symbolic expression which computes $\mathbb{E}[\mathsf{v}]$, namely $\sum_{i=1}^{n} p_i \cdot \sigma_i[\mathsf{v}]$, as $\sigma_i[\mathsf{v}]$ is the symbolic expression which is the value of v along i^{th} path. In terms of Algorithm 1, we use Algorithm 4 for AddEncoding on line 36. Then, for Solve, on line 37, we use an automated decision procedure to solve the query

$$\forall \alpha_1, \ldots, \alpha_n \,.\, Enc_a \bowtie f(\alpha_1, \ldots, \alpha_n)$$

where, as before, $\alpha_1, \ldots, \alpha_n$ are the universal symbolic variables representing the input variables of Prog.

3.4 Formalization

In this section we present the formalization of our method. First, we will present our notation and definitions (§ 3.4.1), and then state a series of soundness theorems for our technique (§ 3.4.2). The goal of this section is to describe how the symbolic state used by Algorithm 1, $R = (\sigma, \varphi, P)$, is an abstraction of a *distribution of program memories*.

3.4.1 Notation & Definitions. To begin, we will define some notation:

- Let $\mathcal{V}$ be the set of all values, $\mathcal{X}$ be the set of all program variables, $\mathcal{Z}_U$ be the set of all universal symbolic variables, $\mathcal{Z}_P$ be the set of all probabilistic symbolic variables, and $\mathcal{Z} = \mathcal{Z}_U \cup \mathcal{Z}_P$ be the combined set of all symbolic variables.
- Let $a_\forall : \mathcal{Z}_U \to \mathcal{V}$ be an assignment of universal symbolic variables to values and let $A_\forall$ be the set of all such assignments.
- Similarly, let $a_p : \mathcal{Z}_P \to \mathcal{V}$ be an assignment of probabilistic symbolic variables to values and let A_p be the set of all such assignments.

- Let $m : \mathcal{X} \to \mathcal{V}$ be a program memory which translates program variables into values, and let M be the set of all program memories.
- Let $d : M \to (\mathcal{V} \to [0,1])$ be a distribution expression parameterized by program memories, and let *Dists* be the set of all distribution expressions.
- Let $\mu : A_\forall \times M \to [0,1]$ be a distribution of program memories parameterized by assignments to universal symbolic variables and let $Dists_M$ be the set of all parameterized distributions of program memories.

Additionally, we will use emphatic brackets ($[\![\cdot]\!]$) for two purposes:

- If e is a *program* expression containing the program variables $\mathsf{x}_1, \ldots, \mathsf{x}_n \in \mathcal{X}$, and $m \in M$, then
$$[\![e]\!]m = \mathsf{eval}(e[\mathsf{x}_1 \mapsto m(\mathsf{x}_1), \ldots, \mathsf{x}_n \mapsto m(\mathsf{x}_n)])$$
- If e is a *symbolic* expression containing the symbolic variables $\alpha_1, \ldots, \alpha_n \in \mathcal{Z}_U$ and $\delta_1, \ldots, \delta_m \in \mathcal{Z}_P$, and $a_\forall \in A_\forall$ and $a_p \in A_p$, then
$$[\![e]\!]a_\forall a_p = \mathsf{eval}(e[\alpha_1 \mapsto a_\forall(\alpha_1), \ldots, \alpha_n \mapsto a_\forall(\alpha_n), \delta_1 \mapsto a_p(\delta_1), \ldots, \delta_m \mapsto a_p(\delta_m)])$$

With this notation in hand, we can now define what it means for R to be an abstraction of a distribution of programs memories.

Definition 3.1. Let $R = (\sigma, \varphi, P)$ be the abstraction generated by Algorithm 1 where $\varphi : A_\forall \times A_p \to \{0,1\}$ denotes whether a path condition is true or false under the given assignments, $\sigma : \mathcal{X} \to \mathit{SymExprs}$ is a mapping from program variables to symbolic expressions generated through symbolic execution, and $P : A_\forall \to \mathcal{Z}_P \rightharpoonup (\mathcal{V} \to [0,1])$ is a mapping from probabilistic symbolic variables to the distribution it is sampled from, parameterized by assignments to universal symbolic variables. Additionally, for every assignment of universal symbolic variables, $a_\forall \in A_\forall$, $\mathrm{dom}\,(P(a_\forall)) = \{\delta_1, \ldots, \delta_k\}$. Let $\alpha_1, \ldots, \alpha_l \in \mathcal{Z}_U$ be the universal symbolic variables which correspond to the l parameters to the program. For every assignment of probabilistic and universal symbolic variables, $a_\forall \in A_\forall, a_p \in A_p$, let $\nu : A_\forall \to (A_p \to [0,1])$ be a distribution of assignments to probabilistic symbolic variables parameterized by assignments to universal symbolic variables, defined as
$$\nu(a_\forall, a_p) \triangleq \prod_{i=1}^{k} \Pr_{v \sim P(a_\forall, \delta_i)}[v = a_p(\delta_i)].$$

We say that a distribution μ satisfies our abstraction R if, for all assignments to universal symbolic variables, $a_\forall \in A_\forall$, $\Pr_{a'_p \sim \nu(a_\forall)}[\varphi(a_\forall, a'_p) = 1] > 0$, and if $\nu_\varphi : A_\forall \to (A_p \to [0,1])$ is defined as
$$\nu_\varphi(a_\forall, a_p) \triangleq \frac{\Pr_{a'_p \sim \nu(a_\forall)}[a'_p = a_p \wedge \varphi(a_\forall, a'_p) = 1]}{\Pr_{a'_p \sim \nu(a_\forall)}[\varphi(a_\forall, a'_p) = 1]}.$$

Additionally, define $\mathsf{toMem} : (\mathcal{X} \to \mathit{SymExprs}) \to A_\forall \to A_p \to M$ as
$$\mathsf{toMem}(\sigma, a_\forall, a_p) \triangleq \lambda(\mathsf{x} : \mathcal{X})\,.\,[\![\sigma(\mathsf{x})]\!]a_\forall a_p,$$
and let $\mathsf{fromMem}(\sigma, a_\forall, m) = (\mathsf{toMem}(\sigma, a_\forall))^{-1}(m)$. Then,
$$\mu(a_\forall, m) \triangleq \sum_{a_p \in \mathsf{fromMem}(\sigma, a_\forall, m)} \nu_\varphi(a_\forall, a_p).$$

3.4.2 Soundness Theorems. Note that in Algorithm 1 we represent the symbolic program state as a five-tuple, namely $S = (I, \varphi, \sigma, P, p)$; however, in our abstraction R, we omit I and p. I represents the current executing instruction which we remove as we prove soundness locally for each type of statement in **pWhile**, rather than prove soundness for the entirety of Algorithm 1. Therefore, there is no need to track what the current instruction is. For p, we will begin by proving that p can be completely reconstructed using φ and P and so there is no need to track it separately.

Theorem 3.2 (Equivalency between p and (φ, P)). *For all program statements, S in **pWhile**, Algorithm 1 maintains*

$$p = \frac{\sum_{v_1,\ldots,v_n \in \mathcal{D}} [\varphi\{\delta_1 \mapsto v_1, \ldots, \delta_n \mapsto v_n\}]}{|\mathcal{D}|}$$

where $\{\delta_1, \ldots, \delta_n\} = \mathrm{dom}\,(P)$, or all the probabilistic symbolic variables in the program, and $\mathcal{D} = \mathrm{dom}\,(P[\delta_1]) \times \cdots \times \mathrm{dom}\,(P[\delta_n])$, or all the assignments to the probabilistic symbolic variables $\delta_1, \ldots, \delta_n$.

We will now prove soundness for Algorithm 1 by showing that our abstraction R respects the semantics defined in the supplemental material for each of the three main types of statements in **pWhile**, namely *assignment*, *sampling*, and *branching*. A proof for each theorem can be found in the supplemental material.

Theorem 3.3 (Correctness of Assignment (Lines 13 to 14)). *If a distribution μ satisfies $R = (\varphi, \sigma, P)$, and Algorithm 1 reaches an assignment statement of the form $\mathsf{x} \leftarrow e$, then $\mu_{\mathsf{x} \leftarrow e}$, the distribution of program memories after executing $\mathsf{x} \leftarrow e$, as defined in the supplemental material, satisfies $R' = (\varphi, \sigma', P)$, which is produced by Algorithm 1.*

Theorem 3.4 (Correctness of Sampling (Lines 18 to 19)). *If a distribution μ satisfies $R = (\varphi, \sigma, P)$ and Algorithm 1 reaches a sampling statement of the form $\mathsf{x} \sim d$, then $\mu_{\mathsf{x} \sim d}$, the distribution of program memories after executing $\mathsf{x} \sim d$, as defined in the supplemental material, satisfies $R' = (\varphi, \sigma', P')$, which is produced by Algorithm 1.*

Theorem 3.5 (Correctness of Branching (Lines 24, 27 and 28)). *If a distribution μ satisfies $R = (\varphi, \sigma, P)$, and Algorithm 1 encounters a branching statement of the form* **if** c **then goto** T*, then μ_c satisfies $R_{true} = (\varphi \wedge c_{sym}, \sigma, P)$ and $\mu_{\neg c}$, the distribution of program memories after executing* **if** c **then goto** T*, as defined in the supplemental material, satisfies $R_{false} = (\varphi \wedge \neg c_{sym}, \sigma, P)$. Additionally, for all $a_\forall \in A_\forall$,*

$$\sum_{m \in \{m \in M \mid [\![c]\!]m = 1\}} \mu(a_\forall, m) = [\![p_c]\!]a_\forall \quad \textit{and} \quad \sum_{m \in \{m \in M \mid [\![c]\!]m = 0\}} \mu(a_\forall, m) = [\![p'_c]\!]a_\forall.$$

4 CASE STUDIES

In this section we will briefly explain each of the case studies that we use in our evaluation (§ 5). For each case study, we will explain, (1) the algorithm, (2) the property we aim to verify using our technique, and (3) any variants to the core algorithm we consider. For the sake of presentation, we present the queries using the program variable names instead of the equivalent symbolic variable representations.

4.1 Freivalds' Algorithm

Background. Freivalds' algorithm [Freivalds 1977] is a randomized algorithm used to verify matrix multiplication in $O(n^2)$ time. Given three $n \times n$ matrices A, B, and C, Freivalds' algorithm

Algorithm 5 Freivalds' Algorithm

Require: A, B, and C are $n \times n$ matrices
1: **function** FREIVALDS(A, B, C, n)
2: $\vec{r} \leftarrow$ An empty $n \times 1$ vector
3: **for** $i \leftarrow 1, n$ **do**
4: $\vec{r}[i] \sim \mathsf{UniformInt}(0, 1)$
5: $\vec{D} \leftarrow A \times (B\vec{r}) - C\vec{r}$
6: **return** $\vec{D} = \begin{pmatrix} 0 & \cdots & 0 \end{pmatrix}^{\mathrm{T}}$

checks whether $A \times B = C$ by generating a random $n \times 1$ vector containing 0s and 1s, $\vec{r}$ and checks whether $A \times (B\vec{r}) - C\vec{r} = (0, \ldots, 0)^T$. If so, the algorithm returns `true`, and `false` otherwise. If $A \times B = C$, the algorithm returns `true` with probability 1 (always). However, if $A \times B \neq C$, the probability that the algorithm returns `true` is at most $\frac{1}{2}$. Pseudocode for the algorithm is presented in Algorithm 5.

Target queries. We want to verify that the false-positive error rate does not exceed $\frac{1}{2}$. Letting $\psi := A \times B \neq C \wedge \text{FREIVALDS}(A, B, C, n) = \texttt{true}$, our query is

$$\forall A, B, C, n \;.\; Enc_{\psi} \leq \frac{1}{2}. \tag{2}$$

Algorithm 6 Freivalds' Algorithm (Multiple)

1: **function** MULTFREIVALDS(A, B, C, n, k)
2: **for** $i \leftarrow 1, k$ **do**
3: **if** FREIVALDS(A, B, C, n) = `false` **then**
4: **return** `false`
5: **return** `true`

To reduce the probability of false-positive in Algorithm 5 can be run k times and returns `true` only if each call to Freivalds' algorithm returns `true`, and `false` otherwise. The pseudocode for this variant is presented in Algorithm 6. Since each trial is independent, the probability of a false-positive given that $A \times B \neq C$ is at most $\left(\frac{1}{2}\right)^k$. Thus, letting $\psi := A \times B \neq C \wedge \text{MULTFREIVALDS}(A, B, C, n, k) =$ `true`, our query becomes:

$$\forall A, B, C, n, k \;.\; Enc_{\psi} \leq \left(\frac{1}{2}\right)^k.$$

4.2 Reservoir Sampling

Background. Reservoir sampling [Vitter 1985] is an online, randomized algorithm to get a simple random sample of k elements from a population of n elements. It uses uniform integer samples to constantly maintain a set of k elements drawn from the set of n elements. For the full algorithm, see Algorithm 7. We want to verify that each possible sample is returned with equal probability, namely $1/\binom{n}{k}$ or, in other words, the output distribution of samples is uniform. To do this, we instead prove an equivalent property that if all elements of A are distinct then the probability that any element of A appears in S is k/n.

Algorithm 7 Reservoir Sampling

Require: $1 \le k \le n$

```
function ReservoirSampling(A[1..n], k)
    S ← An empty list of size k
    for i ← 1, k do
        S[i] ← A[i]
    for i ← k + 1, n do
        j ← UniformInt(1, i)
        if j ≤ k then
            S[j] ← A[i]
    return S
```

Target query. One way of framing the query is to ask the probability of the first element of A appearing in S. If all elements of A are distinct, then this probability will be the same for all elements in A. Therefore, let $\psi := A[1] \neq \cdots \neq A[n] \wedge A[1] \in \textsc{ReservoirSampling}(A, k)$. Then we can take the following query:

$$\forall A, n, k \;.\; Enc_{\psi} = \frac{k}{n}.$$

4.3 Randomized Monotonicity Testing

Algorithm 8 Randomized Monotonicity Testing [Goldreich 2017]

```
function MonotoneTest(f, n)
    l ← ⌈log_2 n⌉, a ← 1, b ← n
    i ← UniformInt(1, n)
    for t ← 1, l do
        p ← ⌈a + b/2⌉
        if i ≤ p then
            if f[i] > f[p] then
                return false
            b ← p
        else
            if f[i] < f[p] then
                return false
            a ← p + 1
    return true
```

Background. Property testing is a subfield of computer science which aims to develop algorithms to determine whether a mathematical object has some property. For example, given a function $f : [n] \to R_n$ where $[n] = \{1, \ldots, n\}$ and R is some ordered set, how can we determine whether f is monotone? That is, can we check that $f(i) \le f(j)$ whenever $i \le j$? Algorithm 8 is a randomized algorithm inspired by the binary search algorithm to rapidly determine how "far" from monotone f is [Goldreich 2017].

To get an intuitive understanding of this algorithm suppose that f is an array of size n where each element is distinct. Algorithm 8 selects an element $i \in [1, n]$ at random and attempts to find the value $f(i)$ in the array using binary search. If f is actually (strictly) monotone, then it will find

$f(i)$ at location i, and will accept if and only if $f(i)$ is found during this search at all. However, if f is not monotone, specifically δ-*far* from monotone, it will reject with probability greater than δ [Goldreich 2017], where δ-far means that $\delta \cdot n$ points need to be changed in f in order to make f monotone. This is the property that we want to check.

Let DistToMono(f) be a function which returns the number of elements whose order needs to be changed in order to make f monotone. Letting $\psi := \text{DistToMono}(f) = k \wedge \textsc{MonotoneTest}(f, n) =$ `false`, we take the query

$$\forall f, n \,.\, Enc_{\psi} > \frac{k}{n}.$$

4.4 Randomized Quicksort

Algorithm 9 QuickSort

```
1:  function QuickSort(A, p, r)
2:      if p < r then
3:          q ← Partition(A, p, r)
4:          QuickSort(A, p, q − 1)
5:          QuickSort(A, q + 1, r)

6:  function Partition(A, p, r)
7:      i ← UniformInt(p, r)
8:      exchange A[r] with A[i]
9:      x ← A[r]
10:     i ← p − 1
11:     for j ← p, r − 1 do
12:         if A[j] ≤ x then
13:             i ← i + 1
14:             exchange A[i] with A[j]
15:     exchange A[i + 1] with A[r]
16:     return i + 1
```

Background. Quicksort is a popular sorting algorithm which uses partitioning in order to achieve efficient sorting. While there are many ways of choosing a pivot element, one way is through a uniform random sample. Using this pivot method, the *expected* number of comparisons required is $1.386n \log_2(n)$ where n is the length of the array.

Target query. We use the method described in § 3.3.1 to compute $\mathbb{E}$ [`num_comps`], where `num_comps` is a program variable which is initially set to 0 and is incremented on lines 2 and 12. Intuitively, `num_comps` counts the number of comparisons performed on a given run of QuickSort.

4.5 Interlude: Hash Functions

The remaining case-studies make use of *hash functions* in order to achieve their guarantees. For non-cryptographic purposes, hash functions allow for rapid indexing into data structures and in many cases allow for constant time data retrieval. Formally, hash functions take elements from some universe and map them to a fixed range, often between $[1, n]$ where n is the length of an array. Ideally, these functions distribute elements from the universe *uniformly* across the range, so distinct elements hash to the same target with low probability. If h is a hash function, and x, y are

Algorithm 10 Uniform Hash Function

```
1:  function HashCreate(min, max)
2:      H ← An empty, uninitialized hash function
3:      H.Map ← An empty map
4:      H.min ← min
5:      H.max ← max
6:      return H

7:  function Hash(H, x)
8:      if x ∈ H.Map then
9:          return H.Map[x]
10:     else
11:         i ← UniformInt(H.min, H.max)
12:         H.Map[x] ← i
13:         return i
```

arbitrary elements, we define a *hash collision* to be when $h(x) = h(y)$ where $x \neq y$. In the upcoming examples, the probability of a certain number of hash collisions occurring is directly connected to the error rate of the examples.

To properly analyze programs which employ hashing we consider ideal, *uniform* hash functions. Therefore, we model hash functions through uniform random samples. Pseudocode for how we model hash functions is provided in the function HashCreate in Algorithm 10, Line 1 Essentially, to hash an unseen element, x, we randomly sample an integer between $H.min$ and $H.max$. We then record that sample in $H.Map$ in order to maintain determinism. To hash x again in the future, we simply retrieve the sample from the mapping. This ensures that each element is hashed uniformly and independently the first time, while subsequent hashes of an element always return the same result.

4.6 Bloom Filter

Background. A Bloom filter [Bloom 1970] is a space-efficient, probabilistic data structure used to rapidly determine whether an element is in a set. Fundamentally, it is a bit-array of n bits and k associated hash functions, each of which maps elements in the set to places in the bit-array. To insert an element, x, into the filter, x is hashed using each of the k hash functions to get k array positions. All of the bits at these positions are set to 1. To check whether an element y is in the filter, y is again hashed by each of the hash functions to get k array positions. The bits at each of these positions are checked and the filter reports that y is in the filter if, and only if, each of the bits are set to 1. Note that false-positives are possible due to hash collisions, but false negatives are not. Pseudocode for a Bloom filter is provided in Algorithm 11.

Target query. In order to bound the false-positive error rate, most implementations of Bloom filters take in the expected number of elements to be inserted as well as the *desired* false-positive error rate. From these two quantities, the optimal size of the bit-array, n, as well as the number of hash functions, k can be computed. If m is the expected number of elements and ε is the desired error rate, then the optimal settings of n and k are

$$n = -\frac{m \ln \varepsilon}{(\ln 2)^2} \quad \text{and} \quad k = -\frac{\ln \varepsilon}{\ln 2} \tag{3}$$

Algorithm 11 Bloom Filter

```
1:  function BloomCreate(m, ε)
2:      B ← An empty, uninitialized Bloom Filter
3:      B.Arr ← A bit-array of size −m ln ε / (ln 2)^2
4:      B.H ← A list of −ln ε / ln 2 independent hash functions
5:      return B

6:  function BloomInsert(B, x)
7:      for all h ∈ B.H do
8:          B.Arr[h(x)] ← 1
9:      return B

10: function BloomCheck(B, x)
11:     for all h ∈ B.H do
12:         if ¬B.Arr[h(x)] then
13:             return false
14:     return true

Require: X[1] ≠ ··· ≠ X[m] ≠ y
15: function Main(X, y, m, ε)
16:     B ← BloomCreate(m, ε)
17:     for i ← 1, m do
18:         B ← BloomInsert(B, X[i])
19:     return BloomCheck(B, y)
```

We want to prove that for a given m and ε that the actual false-positive rate does not exceed ε. In other words, we want to show that the probability that Main (on line 15 of Algorithm 11) returns `true` does not exceed ϵ.

To capture this property, we let $\psi := X[1] \neq \cdots \neq X[m] \neq y \wedge \textsc{Main}(X, y, m, \epsilon) = \texttt{true}$ and take the query:

$$\forall X, y, m, \epsilon \;.\; Enc_{\psi} \leq \epsilon$$

As we show later, setting n and k based on the number of expected elements m and desired false-positive error rate, ϵ, as in Equation (3) does *not* guarantee that the actual false-positive error rate is ϵ. In actuality, ϵ will only be a lower bound for the actual false-positive error rate.

4.7 Count-min Sketch

Background. A Count-min Sketch [Cormode and Muthukrishnan 2005] is another space-efficient, probabilistic data structure which is used as a frequency table for a stream of data. It constantly maintains an estimate for the number of times an element x has been seen while using sub-linear space. Behind the scenes, the data structure is a 2D array of w columns and d rows, and d pairwise independent hash functions, one per row. To create a new count-min sketch, ϵ and γ are taken as arguments representing the additive factor for the count to be off and the probability of error occurring, respectively. Most commonly, $w = \lceil \frac{e}{\epsilon} \rceil$ and $d = \lceil \ln \frac{1}{\gamma} \rceil$. To update the count for an element x, for each row j, x is hashed using j's hash function, h_j, to get the column index k. That is, $k = h_j(x)$. The value at row j and column k is then updated. Let *sketch* be the 2D array. Then, to get the count for x, $\hat{a}_x$, x is hashed using each row's associated hash function to index into

Algorithm 12 Count-min Sketch

```
1:  function SketchCreate(ϵ, γ)
2:      w ← ⌈e/ϵ⌉
3:      d ← ⌈ln 1/γ⌉
4:      C ← An empty, uninitialized count-min sketch
5:      C.sketch ← An empty w × d array
6:      C.H ← A list of d independent hash functions
7:      return C

8:  function SketchUpdate(C, x)
9:      for j ← 1, d do
10:         C.sketch[j, C.H[j](x)] ← C.sketch[j, C.H[j](x)] + 1
11:     return C

12: function SketchEstimate(C, x)
13:     â_x ← ∞
14:     for j ← 1, d do
15:         â_x ← min(â_x, C.sketch[j, C.H[j](x)])
16:     return â_x

17: function Main(ϵ, γ, X, n)
18:     C ← SketchCreate(ϵ, γ)
19:     for i ← 1, n do
20:         C ← SketchUpdate(C, X[i])
21:     return SketchEstimate(C, X[1]) > 1 + ϵn
```

sketch and the minimum element among each of the d counts is chosen as the estimate, or, in other words, $\hat{a}_x = \min_j sketch[j, h_j(x)]$. This count has the property that $\hat{a}_x \leq a_x + \epsilon N$ with probability $1 - \gamma$, where a_x is the "true" count for x and $N = \sum_x a_x$, or the total number of elements seen by the sketch. This is the property we wish to verify. Pseudocode for count-min sketch is provided in Algorithm 12.

Target query. We want to prove that for all ϵ and γ, the probability of Main (line 17 of Algorithm 12) returning `true` does not exceed γ. To capture this property, we let $\psi := X[1] \neq \cdots \neq X[n] \wedge \textsc{Main}(\epsilon, \gamma, X, n)$ and take the following query:

$$\forall \epsilon, \gamma, X, n \;.\; Enc_{\psi} \leq \gamma$$

5 IMPLEMENTATION AND EVALUATION

5.1 Implementation

We have implemented a prototype of our tool, Plinko, on top of KLEE [Cadar et al. 2008], a robust symbolic execution engine. Plinko implements the *probabilistic symbolic* execution algorithm as described in Algorithm 1. For the purpose of our implementation, we modify KLEE to support creation of *probabilistic symbolic* variables as described in § 3.2 whose values can be sampled from a distribution (δ) by invoking PSESample as shown in the ***sampling*** statement [Algorithm 1, Line 16]. We use the *path constraints* (φ) from the *state* data-structure (maintained by KLEE) at each of the

assignment [Algorithm 1, Line 11] and ***branch*** [Algorithm 1, Line 22] statements to get the final encodings Enc_ψ and Enc_a.

For solving, Plinko uses Z3 [de Moura and Bjørner 2008], a popular SMT solver, to either prove or refute our queries. Depending on the type of query, whether it be a probability bound query or an expected value computation, we construct the query in the SMTLIB2 [Barrett et al. 2016]. All the universal symbolic variables are then universally quantified in the query and Enc_ψ is converted into the SMTLIB language. We then pass the converted query to Z3 which either verifies the query or generates a counterexample which disproves the claim.

5.2 Evaluation

To evaluate Plinko, we answer the following questions:

- (***Q1***) How efficiently can Plinko prove probabilistic properties on complex programs?
- (***Q2***) How does the form of the query affect performance?
- (***Q3***) How does the domain of the universal symbolic variables affect performance?
- (***Q4***) Can Plinko find bugs in probabilistic programs?

To answer (***Q1***), we implement, or collect open-source implementations, for each of the case studies presented in § 4 in C++, run them on Plinko, and record a variety of performance metrics, such as timing, number of paths explored, etc. We note that we used publicly available implementations for both of the data structures we present, namely Bloom Filter[1] and Count-min Sketch[2] and adapted them to be analyzed by Plinko and to use our idealized hash functions as presented in § 4.5. In certain cases, we "concretize" some input parameters, often the sizes of the input arrays, in order to constrain the paths explored by the symbolic execution engine. It was required in our setting, as contrary to most applications of symbolic execution that attempt to test the program, we attempt to *verify* these examples in our evaluation. This necessitates exhaustive path enumeration within a reasonable time. We selected our parameter settings (table 1, last column) such that a complete path enumeration, and query verification, could be carried out within 10 minutes. Note that when an array length is concretized, the array contents still remain symbolic. For example, for Quicksort, if the array length is concretized to 4, Plinko searches over all possible arrays of size 4. There is no conceptual obstacle to using our method without concretization if we are interested in bug-finding; Plinko produces a sound lower-bound on exploring a subset of paths (see § 3.3). However, implementation difficulties exist; for instance, there is currently limited support for running KLEE without concretizing array lengths.

To answer (***Q2***) and (***Q3***), we analyze how changes to the query for Freivalds' Algorithm (§ 4.1) affect performance. In general, these changes narrow the domain of possible assignments to the universal symbolic variables that Z3 must consider, thus allowing Z3 to arrive at a conclusion faster. We also discuss how narrowing this space weakens the guarantees that Plinko can provide.

For (***Q4***), we consider two types of bugs: implementation bugs where programmer errors result in the query being falsified, and specification bugs, where the specification is incorrect. We find that we can effectively find, and address, both of these bugs using the results Plinko provides.

We conduct our experiments a machine equipped with an Intel Core i7-5820K running at 3.3 GHz and 32 GB of RAM running Arch Linux with kernel 5.12.14. For each experiment, we report the following metrics: the amount of time taken by Plinko to explore the paths of the program and to generate the query, the amount of time Z3 takes to solve the query, the total amount of time taken, the number of lines in the C++ code, the number of paths Plinko explored, the number of

[1] https://github.com/jvirkki/libbloom
[2] https://github.com/alabid/countminsketch

Table 1. Performance metrics for each of the case studies presented in § 4.

	Timing (sec.)						
Case Study	**KLEE**	**Z3**	**Total**	**Lines**	**Paths**	**Samples**	**Concretizations**
Freivalds'	5	70	**75**	97	8	2	$n = 2$
Freivalds' (Multiple)	44	255	**299**	96	8	21	$(n, k) = (3, 7)$
Reservoir Sampling	49	79	**128**	52	63	5	$(n, k) = (10, 5)$
Monotone Testing	4	352	**356**	69	30	1	$n = 23$
QuickSort	95	489	**584**	65	120	10	$n = 5$
Bloom Filter	256	231	**487**	386	83	8	$(m, \epsilon) = (3, 0.4)$
Count-min Sketch	108	206	**314**	245	2	8	$(n, \epsilon, \gamma) = (4, 0.5, 0.25)$

Table 2. Performance metrics of four different ways of specifying $A \times B \neq C$ in the query for Freivalds' Algorithm (§ 4.1). Here, $n = 2$ and all elements of the matrices are C++ `ints`.

	Timing (sec.)			
Spec. for $A \times B \neq C$	**KLEE**	**Z3**	**Total**	**Paths**
AllOff	2	1	**3**	2
SomeOff	5	70	**75**	8
FirstOff	1	5	**6**	2
OneOff	2	17	**19**	2

random samples, and the maximum value "concretized" universal symbolic variables could take such that the query could be proven (or disproven) in the time allotted.

***(Q1)** Discussion.* Table 1 summarizes our experimental results from running standard versions for each of the case studies presented in § 4. All queries, except for the one for Bloom filter, which we will discuss later, were proven to be true within ten minutes. As far as we know, *none* of these examples can be verified by existing automated tools; our method is the first to handle these examples fully automatically.

We begin by making a few general observations. First, the presented examples showcase a range of path counts and number of random samples. For instance, even when we restricted the array to be of length 5, we still explored and reasoned over 120 paths and 10 probabilistic symbolic variables when analyzine QuickSort. Second, the time Plinko takes to execute Algorithm 1 is usually much shorter than the time it takes for Z3 to solve the query; this suggests that constraint solving is the main bottleneck in our approach, not path exploration.

***(Q2)** Discussion.* To better understand Plinko's performance we consider how changes to the phrasing of the query for Freivalds' Algorithm affects performance. Recall that Freivalds' Algorithm efficiently determines whether $A \times B = C$, where A, B, and C are $n \times n$ matrices; however, it has a false-positive error rate of at most $\frac{1}{2}$. In Equation (2) we set $\psi := A \times B \neq C \wedge \textsc{Freivalds}(A, B, C, n) = \texttt{true}$ but did not state exactly how $A \times B$ must differ from C.

Table 3. Performance metrics for four different domains from which the elements of A, B, and C are drawn from in the implementation for Freivalds' Algorithm (§ 4.1). We only consider 2×2 matrices for each data type and use SomeOff (Equation (4)) to specify that $A \times B \neq C$. In each variant, Plinko explored 8 paths.

	Timing (sec.)		
Data Type	**KLEE**	**Z3**	**Total**
`long int`	11	1,589	**1,600**
`int`	5	70	**75**
`short int`	3	7	**10**
`char`	2	1	**3**

We consider four different ways of specifying that $A \times B \neq C$:

$$\textsc{AllOff} = \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} (A \times B)_{i,j} \neq C_{i,j} \qquad \textsc{SomeOff} = \bigvee_{i=1}^{n} \bigvee_{j=1}^{n} (A \times B)_{i,j} \neq C_{i,j} \tag{4}$$

$$\textsc{FirstOff} = (A \times B)_{0,0} \neq C_{0,0} \qquad \textsc{OneOff} = \exists i, j \,.\, (A \times B)_{i,j} \neq C_{i,j}$$

AllOff states that *every* element of $A \times B$ must be different from the corresponding element in C, whereas SomeOff only states that *at least one* element be deifferent between $A \times B$ and C. Similarly, FirstOff states that the first element of each matrix be different and OneOff states that *an* element of $A \times B$ must differ from C. For OneOff, we represent i and j as universal symbolic variables and universally quantify them in Equation (2). In all four cases, we only consider 2×2 matrices which contain C++ `int`s and present the performance results in Table 2. We quickly note that Plinko explored eight paths for the SomeOff specifcation compared to the two paths for each of the other three variants. Since specifications are written in the source program itself the complexity of said program can increase or decrease depending on the definition of ψ. In the case of SomeOff, Plinko explores more paths than the other variants as SomeOff is the only variant which does not provide a concrete number for the number of differing elements between the two matrices.

In general, the results in Table 2 suggest that simpler and more specific settings of ψ result in increased performance. The strongest, and best performing specification was AllOff, whereas the most general, and the least performing specification was SomeOff. Additionally, FirstOff performed considerably better than OneOff, while being similar. Intuitively, this makes sense as AllOff drastically limits the amount of possible matrices that Z3 has to consider, whereas SomeOff requires Z3 to reason about *all* matrices A, B, and C such that $A \times B \neq C$. The same reasoning can be applied to the relationship between FirstOff and OneOff; OneOff already considers whether FirstOff holds. Therefore, while performing the worst, SomeOff provides the strongest guarantee out of all the other variants, followed by OneOff, then FirstOff, and then finally, AllOff.

***(Q3)** Discussion.* We additionally consider how the size of the domain that Z3 has to reason over impacts performance. This domain is primarily determined by the declared datatype of the C++ variables. We test four variants of Freivalds' Algorithm where we change the data type for the elements of the three matrices. For each data type, we restrict ourselves to 2×2 matrices and specify that $A \times B \neq C$ using SomeOff (Equation (4)). The performance results are presented in Table 3. On the evaluating machine, `long int`s are eight bytes, `int`s are four bytes, `short int`s are two bytes, and `char`s are a single byte.

As suspected, matrices containing `char`s performed the best, completing the verification in only 3.53 seconds whereas matrices containing `long int`s took over 26 minutes to complete. On the

other hand, using `chars` provides the weakest guarantee whereas `long ints` provide the strongest. For many applications, however, analyzing variants of the program with smaller data types might provide sufficient proof of correctness, or already be able to surface bugs.

(Q4) *Discussion.* We were able to find two bugs: one which we seeded in our implementation of Randomized Monotonicity Testing (§ 4.3), and one which we found in our reference implementation of Bloom Filter (§ 4.6). For randomized monotonicity testing, the source material which provides the algorithm and guarantee [Goldreich 2017] presents the algorithm using 1-based indexing. To mimic a common off-by-one error, we did not perform the proper translation during the initialization of l on Line 2 of Algorithm 8. Specifically, we set $l \leftarrow \lceil \log_2 n \rceil$ instead of $l \leftarrow \lceil \log_2(n-1) \rceil$, which is the proper 0-indexed value. This bug increased the probability that a function f would be rejected by Line 2, and so made our queries fail. We found this bug setting $n = 4$. While this was a simple (yet common) off-by-one error, it is a prime example of both how subtle probabilistic reasoning can be and how potentially effective Plinko can be as not only a verification, but also a debugging tool.

The second error we found was in our reference Bloom filter implementation. If we set $m = 3$, or the number of expected elements to be inserted into the Bloom filter, and $\epsilon = 0.4$, or the *desired* false-positive rate, Plinko computes the actual false-positive rate to be $\approx 45.03\%$. Upon closer inspection, we were able to conclude that because approximations must be used in order to compute the number of hash functions and the number of bits needed to achieve ϵ, ϵ can only be a lower bound for the actual false-positive rate. This confirms an observation by Bose et al. [2008], who showed that the commonly reported false-positive rate for the Bloom filter is slightly incorrect.

6 RELATED WORK

Probabilistic Symbolic Execution. Geldenhuys et al. [2012] was the first to propose a method for probabilistic symbolic execution. Given a standard, non-probabilistic program, their technique treats all inputs as drawn from a uniform distribution, and then computes the probabilities of taking particular program paths by using model counting. Later works build on this idea for different applications: analyzing software reliability [Borges et al. 2014; Filieri et al. 2013], quantifying software changes [Filieri et al. 2015], generating performance distribution [Chen et al. 2016], and evaluating worst-case input distributions in network programs [Kang et al. 2021]. More sophisticated schemes apply volume-bound computations instead of model counting, which is a performance bottleneck [Albarghouthi et al. 2017; Sankaranarayanan et al. 2013].

These methods all focus on probabilistic programs where all the program inputs are either known constants, or sampled from a known distribution (often the uniform distribution). In contrast, our technique works for inputs that are unknown values, and not drawn from a known distribution. Applying existing methods for probabilistic symbolic execution to our setting would be prohibitively expensive, since each possible input would need to be considered, and input may range over a very large domain (e.g., integers ranging from -2^{63} to $2^{63} - 1$).

Symbolic inference in probabilistic programs. Probabilistic programming languages are languages enriched with both *sampling* and *conditioning* operations. The combination of these two features allows a probabilistic program to encode a complex distribution; many models of interest in machine learning can be implemented in this way. A recent line of research considers how to perform *inference* on such programs: given an assertion P, what is the probability that P holds in the distribution described by the program? Researchers have considered a variety of approaches, from weighted model counting [Holtzen et al. 2020], to analyzing Bayesian networks [Sampson et al. 2014], to applying computer algebra systems [Claret et al. 2013; Gehr et al. 2016, 2020].

Most existing work on probabilistic programming languages assumes that all quantities in the program are drawn from known distributions—while we consider a family of output distributions,

probabilistic programming languages typically consider a single distribution. It would be interesting to extend our work to handle conditioning, but it would likely be quite challenging since conditioning can transform our symbolic branch probabilities in highly complex ways.

Other automated techniques for probabilistic programs. Automated verification of probabilistic programs is an active area of research. We briefly survey some of the most relevant lines of work.

AxProf [Joshi et al. 2019] is a *statistical testing* approach to analyzing probabilistic programs, which runs the target program multiple times on concrete inputs in order to estimate probabilities and expected values in the output distribution. AxProf is highly efficient, and supports programs with unknown inputs. However, it can only explore a small subset of the input space and cannot provide logical guarantees over all possible inputs.

Probabilistic model checking is a well-studied method for checking logical formulas on probabilistic transition systems [Baier et al. 1997, 2018; Dehnert et al. 2017; Kwiatkowska et al. 2011]. However, probabilistic model checking targets finite state systems, and performance degrades rapidly as the number of states grows. Our technique applies to infinite state systems; our examples, which are finite state, are far too large to encode in existing model checking tools.

Abstract interpretation and *algebraic program analysis* methods have been developed for probabilistic programs [Cousot and Monerau 2012; Wang et al. 2018]. These methods abstract the probabilistic state, losing information in exchange for better performance. In contrast, our method computes path probabilities exactly, in symbolic form.

Finally, there are many *domain-specific* automated analysis for specific probabilistic properties: termination and resource analysis [Chatterjee et al. 2016; Moosbrugger et al. 2021; Wang et al. 2021], accuracy [Chakarov and Sankaranarayanan 2013; Smith et al. 2019], reliability [Carbin et al. 2012], differential privacy [Barthe et al. 2021] and other relational properties [Albarghouthi and Hsu 2018a,b; Farina et al. 2021], and long-run properties of probabilistic loops [Bartocci et al. 2019, 2020]. Our approach aims at a general-purpose analysis.

7 CONCLUSION AND FUTURE DIRECTIONS

We have presented a symbolic execution method for randomized programs, with symbolic variables for modeling unknown inputs and random samples. Going forward, we see two promising directions for further investigation.

Optimizing probabilistic symbolic execution. In this work, we have made little effort to optimize our symbolic execution method. One natural direction is to develop heuristics for exploring paths. It would be interesting to apply techniques from the literature on path prioritization in symbolic execution [Chen et al. 2016; Kang et al. 2021], but perhaps there are other heuristics in the probabilistic setting. For instance, since our method computes path probabilities incrementally, perhaps paths with higher probability mass should be explored first. Another possibility is to develop methods to simplify the path probability expressons; while our method can compute them, the expressions are rather complicated.

Analyzing more complex probabilistic programs. We have evaluated our implementation on relatively standard randomized programs so far, and both KLEE and Z3 support richer programs and hardware features. For instance, Z3 has support for reasoning about floating-point arithmetic; perhaps our symbolic execution method could also be used to validate randomized programs that use floating-point computations. In another direction, recent work develops an extension of KLEE that works on unbounded integers [Kapus et al. 2019]; it could be interesting to see if this technique gives better performance or stronger guarantees when verifying randomized algorithms, which often work with mathematical integers.

REFERENCES

Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. 2017. FairSquare: probabilistic verification of program fairness. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 80:1–80:30. https://doi.org/10.1145/3133904

Aws Albarghouthi and Justin Hsu. 2018a. Constraint-Based Synthesis of Coupling Proofs. In *International Conference on Computer Aided Verification (CAV), Oxford, England.* https://doi.org/10.1007/978-3-319-96145-3_18 arXiv:1804.04052 [cs.PL]

Aws Albarghouthi and Justin Hsu. 2018b. Synthesizing Coupling Proofs of Differential Privacy. *Proceedings of the ACM on Programming Languages* 2, POPL, Article 58 (Jan. 2018). https://doi.org/10.1145/3158146 arXiv:1709.05361 [cs.PL]

Christel Baier, Edmund M. Clarke, Vasiliki Hartonas-Garmhausen, Marta Z. Kwiatkowska, and Mark Ryan. 1997. Symbolic Model Checking for Probabilistic Processes. In *International Colloquium on Automata, Languages and Programming (ICALP), Bologna, Italy (Lecture Notes in Computer Science, Vol. 1256)*. Springer, 430–440. https://doi.org/10.1007/3-540-63165-8_199

Christel Baier, Luca de Alfaro, Vojtech Forejt, and Marta Kwiatkowska. 2018. Model Checking Probabilistic Systems. In *Handbook of Model Checking*. Springer-Verlag, 963–999. https://doi.org/10.1007/978-3-319-10575-8_28

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

Gilles Barthe, Rohit Chadha, Paul Krogmeier, A. Prasad Sistla, and Mahesh Viswanathan. 2021. Deciding accuracy of differential privacy schemes. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. https://doi.org/10.1145/3434289

Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2019. Automatic Generation of Moment-Based Invariants for Prob-Solvable Loops. In *International Symposium on Automated Technology for Verification and Analysis (ATVA), Taipei City, Taiwan (Lecture Notes in Computer Science, Vol. 11781)*, Yu-Fang Chen, Chih-Hong Cheng, and Javier Esparza (Eds.). Springer-Verlag, 255–276. https://doi.org/10.1007/978-3-030-31784-3_15

Ezio Bartocci, Laura Kovács, and Miroslav Stankovic. 2020. Mora – Automatic Generation of Moment-Based Invariants. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Dublin, Ireland (Lecture Notes in Computer Science, Vol. 12078)*, Armin Biere and David Parker (Eds.). Springer-Verlag, 492–498. https://doi.org/10.1007/978-3-030-45190-5_28

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM* 53, 2 (Feb. 2010), 66–75. https://doi.org/10.1145/1646353.1646374

Burton H Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 13, 7 (1970), 422–426.

Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Păsăreanu, and Willem Visser. 2014. Compositional Solution Space Quantification for Probabilistic Software Analysis. *SIGPLAN Not.* 49, 6 (June 2014), 123–132. https://doi.org/10.1145/2666356.2594329

Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel H. M. Smid, and Yihui Tang. 2008. On the false-positive rate of Bloom filters. *Inf. Process. Lett.* 108, 4 (2008), 210–213. https://doi.org/10.1016/j.ipl.2008.05.018

Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, Richard Draves and Robbert van Renesse (Eds.). USENIX Association, 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C Rinard. 2012. Proving acceptability properties of relaxed nondeterministic approximate programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Beijing, China.* 169–180. https://doi.acm.org/10.1145/2254064.2254086

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic program analysis with martingales. In *International Conference on Computer Aided Verification (CAV), Saint Petersburg, Russia.* 511–526. https://www.cs.colorado.edu/~srirams/papers/cav2013-martingales.pdf

Krishnendu Chatterjee, Hongfei Fu, Petr Novotný, and Rouzbeh Hasheminezhad. 2016. Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), Saint Petersburg, Florida.* 327–342. https://doi.org/10.1145/2837614.2837639

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* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/2884781.2884794

Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian Inference Using Data Flow Analysis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 92–102. https://doi.org/10.1145/2491411.2491423

Graham Cormode and S. Muthukrishnan. 2005. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. *J. Algorithms* 55, 1 (April 2005), 58–75. https://doi.org/10.1016/j.jalgor.2003.12.001

Patrick Cousot and Michael Monerau. 2012. Probabilistic Abstract Interpretation. In *European Symposium on Programming (ESOP), Tallinn, Estonia (Lecture Notes in Computer Science, Vol. 7211)*. Springer-Verlag, 169–193. https://www.di.ens.fr/~cousot/publications.www/Cousot-Monerau-ESOP2012-extended.pdf

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. 2017. A storm is Coming: A Modern Probabilistic Model Checker. *CoRR* abs/1702.04311 (2017). arXiv:1702.04311 http://arxiv.org/abs/1702.04311

Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. 2019. Relational Symbolic Execution. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming* (Porto, Portugal) *(PPDP '19)*. Association for Computing Machinery, New York, NY, USA, Article 10, 14 pages. https://doi.org/10.1145/3354166.3354175

Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. 2021. Coupled Relational Symbolic Execution for Differential Privacy. In *European Symposium on Programming (ESOP), Luxembourg City, Luxembourg (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer-Verlag, 207–233. https://doi.org/10.1007/978-3-030-72019-3_8

Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. 2013. Reliability Analysis in Symbolic Pathfinder *(ICSE '13)*. IEEE Press, 622–631.

Antonio Filieri, Corina S. Păsăreanu, and Guowei Yang. 2015. Quantification of Software Changes through Probabilistic Symbolic Execution *(ASE '15)*. IEEE Press, 703–708. https://doi.org/10.1109/ASE.2015.78

Rusins Freivalds. 1977. Probabilistic Machines Can Use Less Running Time. In *Information Processing, Proceedings of the 7th IFIP Congress 1977, Toronto, Canada, August 8-12, 1977*, Bruce Gilchrist (Ed.). North-Holland, 839–842.

Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. Psi: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*. Springer, 62–83.

Timon Gehr, Samuel Steffen, and Martin Vechev. 2020. LambdaPSI: Exact Inference for Higher-Order Probabilistic Programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 883–897. https://doi.org/10.1145/3385412.3386006

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

Oded Goldreich. 2017. *Introduction to Property Testing*. Cambridge University Press. https://doi.org/10.1017/9781108135252

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 140 (Nov. 2020), 31 pages. https://doi.org/10.1145/3428208

Keyur Joshi, Vimuth Fernando, and Sasa Misailovic. 2019. Statistical Algorithmic Profiling for Randomized Approximate Programs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 608–618. https://doi.org/10.1109/ICSE.2019.00071

Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. 2021. Probabilistic Profiling of Stateful Data Planes for Adversarial Testing. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 286–301. https://doi.org/10.1145/3445814.3446764

Timotej Kapus, Martin Nowack, and Cristian Cadar. 2019. Constraints in Dynamic Symbolic Execution: Bitvectors or Integers?. In *Tests and Proofs - 13th International Conference, TAP@FM 2019, Porto, Portugal, October 9-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11823)*, Dirk Beyer and Chantal Keller (Eds.). Springer, 41–54. https://doi.org/10.1007/978-3-030-31157-5_3

Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2013. HAMPI: A Solver for Word Equations over Strings, Regular Expressions, and Context-Free Grammars. *ACM Trans. Softw. Eng. Methodol.* 21, 4, Article 25 (Feb. 2013), 28 pages. https://doi.org/10.1145/2377656.2377662

James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In *International Conference on Computer Aided Verification (CAV), Snowbird, Utah (Lecture Notes in Computer Science, Vol. 6806)*. Springer-Verlag, 585–591.

Marcel Moosbrugger, Ezio Bartocci, Joost-Pieter Katoen, and Laura Kovács. 2021. Automated Termination Analysis of Polynomial Probabilistic Programs. In *European Symposium on Programming (ESOP), Luxembourg City, Luxembourg (Lecture Notes in Computer Science, Vol. 12648)*, Nobuko Yoshida (Ed.). Springer-Verlag, 491–518. https://doi.org/10.1007/978-3-030-72019-3_18

Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and Verifying Probabilistic Assertions. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 112–122. https://doi.org/10.1145/2594291.2594294

Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths. *SIGPLAN Not.* 48, 6 (June 2013), 447–458. https://doi.org/10.1145/2499370.2462179

Raimondas Sasnauskas, Oscar Soria Dustmann, Benjamin Lucien Kaminski, Klaus Wehrle, Carsten Weise, and Stefan Kowalewski. 2011. Scalable Symbolic Execution of Distributed Systems. In *Proceedings of the 2011 31st International Conference on Distributed Computing Systems (ICDCS '11)*. IEEE Computer Society, USA, 333–342. https://doi.org/10.1109/ICDCS.2011.28

Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. 2010. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks before Deployment. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks* (Stockholm, Sweden) *(IPSN '10)*. Association for Computing Machinery, New York, NY, USA, 186–196. https://doi.org/10.1145/1791212.1791235

Steve Selvin. 1975. Letters to the Editor. *The American Statistician* 29, 1 (1975), 67–71. https://doi.org/10.1080/00031305.1975.10479121 arXiv:https://doi.org/10.1080/00031305.1975.10479121

Calvin Smith, Justin Hsu, and Aws Albarghouthi. 2019. Trace Abstraction modulo Probability. *Proceedings of the ACM on Programming Languages* 3, POPL, Article 39 (Jan. 2019). https://doi.org/10.1145/3290352 arXiv:1810.12396 [cs.PL]

Jeffrey S. Vitter. 1985. Random Sampling with a Reservoir. *ACM Trans. Math. Softw.* 11, 1 (March 1985), 37–57. https://doi.org/10.1145/3147.3165

Di Wang, Jan Hoffmann, and Thomas Reps. 2021. Central Moment Analysis for Cost Accumulators in Probabilistic Programs. arXiv:2001.10150 [cs.PL]

Di Wang, Jan Hoffmann, and Thomas W. Reps. 2018. PMAF: an algebraic framework for static analysis of probabilistic programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadelphia, Pennsylvania*. 513–528. https://doi.org/10.1145/3192366.3192408