Probabilistic Symbolic Execution

FIRST1 LAST1*, Institution1, Country1
FIRST2 LAST2†, Institution2a, Country2a and Institution2b, Country2b

TODO: Write Abstract

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages.*

Additional Key Words and Phrases: keyword1, keyword2, keyword3

ACM Reference Format:
First1 Last1 and First2 Last2. 2022. Probabilistic Symbolic Execution. *Proc. ACM Program. Lang.* 1, POPL, Article 1 (January 2022), 19 pages.

1 INTRODUCTION

One of the goals of symbolic execution is to identify whether a "bad" state (e.g. an `assert` failure) is reached, signifying a bug in the program. In probabilistic programs, we are often more interested in either whether a "bad" state is reached too often, or, conversely, whether the "good" states are reached often enough.

1.1 Contributions

2 MOTIVATING EXAMPLE

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 gameshow 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 Fig. 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 program property, how do we verify that the program satisfies the property?

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

*with author1 note
†with author2 note

Authors' addresses: First1 Last1, Department1, Institution1, Street1 Address1, City1, State1, Post-Code1, Country1, first1.last1@inst1.edu; First2 Last2, Department2a, Institution2a, Street2a Address2a, City2a, State2a, Post-Code2a, Country2a, first2.last2@inst2a.com, Department2b and Institution2b, Street3b Address2b, City2b, State2b, Post-Code2b, Country2b, first2.last2@inst2b.org.

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

```c
int monty_hall(int choice, bool door_switch) {
        int car_door = uniform_int(1,3);
        int host_door;
        if (choice != 1 && car_door != 1) {
                host_door = 1;
        } else if (choice != 2 && car_door != 2) {
                host_door = 2;
        } else {
                host_door = 3;
        }
        if (door_switch) {
                if (host_door == 1) {
                        if (choice == 2) {
                                choice = 3;
                        } else {
                                choice = 2;
                        }
                } else if (host_door == 2) {
                        if (choice == 1) {
                                choice = 3;
                        } else {
                                choice = 1;
                        }
                } else {
                        if (choice == 1) {
                                choice = 2;
                        } else {
                                choice = 1;
                        }
                }
        }
        if (choice == car_door) {
                return true;
        } else {
                return false;
        }
}
```

Fig. 1. C code for the Monty Hall Problem

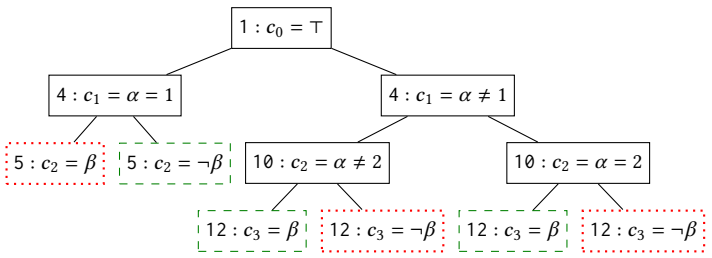


Fig. 2. Execution Tree for the Monty Hall Problem if the car is behind door 1.

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 an execution tree which shows us possible paths through the program.

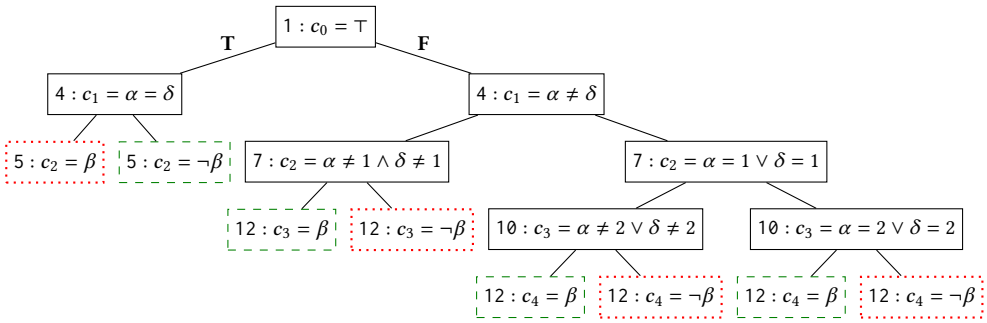


Fig. 3. Execution tree for the Monty Hall Problem with probabilistic symbolic variables.

Suppose we were to run traditional symbolic execution on the program in Fig 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 Fig. ??. For each node in the tree the line number of the branch statement and the path 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`). Symmetric trees can be made for the cases when the car is behind door 2 and door 3.

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 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 also as 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 probabilistic symbolic variable and record the distribution from which the sample is from. We then are able to interpret a branch on a probabilistic symbolic variable as a conditioning operation on the originating distribution.

Now let us return to the Monty Hall problem from Fig. 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 Fig. ??. Note that branches are omitted from the tree if which 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 **T** in Fig. **??**), 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, $\mathbf{T} = ([\alpha = 1] + [\alpha = 2] + [\alpha = 3])/3 = \frac{1}{3}$ as $\delta \in [1, 2, 3]$ and each setting is equally likely. Similarly, $\mathbf{F} = ([\alpha \neq 1] + [\alpha \neq 2] + [\alpha \neq 3])/3 = \frac{2}{3}$.

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 and use automated decision procedures to automate the solving.

3 PROBABILISTIC SYMBOLIC EXECUTION ALGORITHM

In this section, we present our technique for augmenting traditional symbolic execution to support probabilistic programs with discrete sampling instructions. We begin with a short review of a traditional symbolic execution algorithm and then discuss how we calculate exact path probabilities.

3.1 Background

Symbolic execution is a program analysis technique where a program is run on *symbolic* inputs and all program operations are replaced with those which manipulate these symbolic variables. We present a high-level description of how *symbolic execution* works in Alg 6. For each path in the program, a list of *constraints* are stored that encodes the whole path as a *logical* formula. These constraints are over *program* variables that have been marked *symbolic* by the user and are stored in the *state* data structure as *metadata* during the *dynamic* symbolic execution of the program. Apart from *path constraints*, the *state* also contains a mapping of *symbolic* variables to expressions that are either *symbolic* or *concrete* (in the case of concrete execution of the program) and a list of *instructions* currently getting executed as a part of the *state*. During symbolic execution, program state is encoded symbolically in two parts: a conjunctive formula, φ, consisting of the branch conditions which are true for that particular path, called a *path condition*, and a mapping from program variables to symbolic expressions containing constants and *symbolic variables*, σ. Each *instruction* in the program is executed and depending upon the *type* of the *instruction* the *state* along with the *symbolic variables* mapping, (Δ in Alg 6) are updated as described in Algorithm 6. *Symbolic variables* are assigned values by employing an SMT Solver for solving the current set of *constraints* collected so far in the *program path* and also by *concretizing* some of the values that get evaluated as a result of *concrete* execution (e.g. return value from external function calls). When execution reaches an assignment of the form $\mathsf{x} = e$, where e is a constant or program variable, $\sigma[\mathsf{x}]$ is either just the constant, e, or the symbolic representation of e, namely $\sigma[e]$. If e is an expression, we recursively convert each program variable in e to its corresponding symbolic expression found in σ to construct a new symbolic execution, e_{sym}, and set $\sigma[\mathsf{x}] = e_{sym}$. When execution reaches a branch guarded by the condition c, execution proceeds down both branches, one where $\varphi = \varphi \wedge \sigma[c]$, and the other where $\varphi = \varphi \wedge \neg\sigma[c]$.

3.2 Adding Probabilistic Sampling

We present the formalism of our approach on **pWhile**, a core imperative probabilistic programming language, as a model for more general languages:

$$S := \textbf{skip} \mid \mathsf{x} \leftarrow e \mid \mathsf{x} \xleftarrow{\$} d \mid S_1; S_2 \mid \textbf{if } e \textbf{ then } S_1 \textbf{ else } S_2 \mid \textbf{while } e \textbf{ do } S$$

Above, x is a program variable, e is an expression, and d is a *discrete* distribution expression which denotes which distribution the sample should be drawn from. We permit distribution expressions to be (optionally) parameterized by program variables. For example, $\mathsf{UniformInt}(1, \mathsf{x})$ is a uniform

Algorithm 1 Probabilistic Symbolic Execution Algorithm

1: **function** SymbEx(P_{prog} : *Program*)
2: $\varphi_{paths} \leftarrow [], Ex_{stack} \leftarrow [], \Delta \leftarrow [], \phi_{paths} \leftarrow \phi, P \leftarrow \phi$ ▷ Initialization
3: $I_0 \leftarrow$ getStartInstruction(P_{prog})
4: $S_0 \leftarrow [I_0, \phi_{path}, \Delta, \phi]$ ▷ Empty Initial State
5: Ex_{stack}.Push(S_0) ▷ Start with S_{cur} in Execution Stack
6: **while** Ex_{stack} != ϕ **do**
7: $S_{cur} \leftarrow Ex_{stack}$.Pop( ), $I_{cur} \leftarrow S_{cur}[1]$ ▷ Start State, $[I_0, \phi_{path}, \Delta]$
8: **switch** instType(I_{cur}) **do**
9: **case** `make_pse_sym` (x_{var}, d_{dist}) ▷ Create `fresh` Symbolic Variable
10: $I_{cur} \leftarrow$ getNextInstruction(I_{cur})
11: $\phi_{path}, \Delta, P \leftarrow$ **PSESample**($x_{var}, d_{dist}, \phi_{path}, \Delta, P$)
12: $S_{cur} \leftarrow [I_{cur}, \phi_{path}, \Delta, P]$
13: Ex_{stack}.Push(S_{cur})
14: **end case**
15: **case** v := e ▷ Assignment Instruction
16: $I_{cur} \leftarrow$ getNextInstruction(I_{cur})
17: $\phi_{path}, \Delta, P \leftarrow$ **PSEAssignment**(v, e, ϕ_{path}, Δ, P)
18: $S_{cur} \leftarrow [I_{cur}, \phi_{path}, \Delta, P]$
19: Ex_{stack}.Push(S_{cur})
20: **end case**
21: **case if** (c_{sym}) **then** P_1 **else** P_2 ▷ Branch Instruction
22: $Branch_{expr} \leftarrow$ **PSEBranch**($c_{sym}, \phi_{path}, \Delta, P$)
23: $S_{true}, S_{false} \leftarrow$ ForkState (S_{cur}, I_{cur})
24: $I_1 \leftarrow$ getStartInstruction(P_1)
25: $I_2 \leftarrow$ getStartInstruction(P_2)
26: $S_{true} \leftarrow [I_1, \phi_{path} \wedge c_{sym}, \Delta, Branch_{expr}[0]]$
27: $S_{false} \leftarrow [I_2, \phi_{path} \wedge (\neg c_{sym}), \Delta, Branch_{expr}[1]]$
28: Ex_{stack}.Push(S_{false})
29: Ex_{stack}.Push(S_{true}) ▷ Start with True State
30: **end case**
31: **case HALT** ▷ Terminate Instruction
32: $\varphi_{paths} \leftarrow \varphi_{paths}$.Append($\phi_{path}$)
33: **end case**
34: **end switch**
35: **return** φ_{paths}

distribution which selects at random a value between 1 and x (inclusive), where x $\geq$ 1. In order to support sampling instructions we make the following additions to traditional symbolic execution:

- *Probabilistic symbolic variables.* We distinguish two types of symbolic variables: *universal* symbolic variables (identical to those in traditional symbolic execution), and *probabilistic* symbolic variables. For each sampling instruction a new probabilistic symbolic variable is created to denote the result of sample.
- *Distribution map.* We add a new mapping from probabilistic symbolic variables to distribution expressions, P, which 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.* For each path, we adjoin a path probability expression, p, which is parameterized by universal symbolic variables. Now each path can be identified by both its path condition and its probability.

Algorithm 2 PSE Assignment Algorithm

1: **function** PSEAssignment(x, e, φ, σ, P)
2: $\quad e_{sym} \leftarrow \sigma[e]$
3: $\quad \sigma[\mathsf{x}] = e_{sym}$
4: $\quad$ **return** (φ, σ, P)

Assignment. For assignment statements of the form $\mathsf{x} \leftarrow e$, where x is a *program* variable and e is an expression, probabilistic symbolic execution proceeds identically to traditional symbolic execution, as detailed in Alg. 7. On the *symbolic execution* side, the *symbolic variable* mapping (Δ) is updated with the result of the assignment operation is stored back in the current *state* for further execution as shown in Line 17 of Alg 6.

Algorithm 3 PSE Sampling Algorithm

1: **function** PSESample(x, d, φ, σ, P)
2: $\quad \delta \leftarrow$ Generate a fresh probabilistic symbolic variable
3: $\quad \sigma[\mathsf{x}] = \delta$
4: $\quad P[\delta] = d$
5: $\quad$ **return** (φ, σ, P)

Sampling. For sampling statements, $\mathsf{x} \overset{\$}{\leftarrow} d$, where x is a *program* variable and d is a distribution expression, Alg. 8 is used. A fresh probabilistic symbolic variable, δ, is created, σ is updated to be $\sigma[\mathsf{x}] = \delta$, and the original distribution d is recorded in P by setting $P[\delta] = d$. On the *symbolic execution* side, the updated *Distribution Map* (P) with the result of sampling operation is stored back in the current *state* for further execution as shown in Line 11 of Alg 6.

Algorithm 4 PSE Branch Algorithm

1: **function** PSEBranch(c, φ, σ, P)
2: $\quad c_{sym} \leftarrow \sigma[c]$
3: $\quad (\delta_1, \ldots, \delta_n) \leftarrow \mathrm{dom}\,(P)$
4: $\quad (d_1, \ldots, d_n) \leftarrow (P[\delta_1], \ldots, P[\delta_n])$
5: $\quad p_c \leftarrow \dfrac{\sum_{(v_1,\ldots,v_n) \in \mathrm{dom}(d_1)\times\cdots\times\mathrm{dom}(d_n)} [(c_{sym} \wedge \varphi)\{\delta_1 \mapsto v_1, \ldots, \delta_n \mapsto v_n\}]}{\sum_{(v_1,\ldots,v_n) \in \mathrm{dom}(d_1)\times\cdots\times\mathrm{dom}(d_n)} [\varphi\{\delta_1 \mapsto v_1, \ldots, \delta_n \mapsto v_n\}]}$
6: $\quad$ **return** $((\varphi \wedge c_{sym}, \sigma, P, p_c), (\varphi \wedge \neg c_{sym}, \sigma, P, 1 - p_c))$

Branches (Alg. 9). Note that with the inclusion of probabilistic symbolic variables we can now either branch on universal or probabilistic symbolic variables (or both). Intuitively, branches are handled much the same in probabilistic symbolic execution as they are in traditional symbolic execution save for one detail: now branches have a probability associated with them. Given a guard expression c, how do we compute the probability of c being true?

To gain intuition, for now just consider the special case where the guard condition only references universal symbolic variables. Let c_{sym} be the equivalent symbolic expression for the guard c and assume that c_{sym} does not reference any probabilistic symbolic variables. Note that which side of the branch is taken is solely determined by the setting of the univeral symbolic variables. Therefore, one side of the branch must have a probability of 1, and the other side, 0. 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 trickier as given a fixed setting of the universal symbolic variables, it is unclear which branch execution will follow as this is dependent upon the sampling results. Without loss of generality, consider a branch of the form **if** c **then** S_1 **else** S_2. As before, we define $c_{sym} = \sigma[c]$, or the symbolic expression representation of the guard expression, c, and we want to compute the probability of taking the "true" branch and the "false" branch, assuming execution has reached the start of the `if` condition. Since the path condition φ records the necessary constraints on the universal and probabilistic symbolic variables which must hold in order to reach this `if` condition, we can view this probability a *conditional probability*, or the probability that c_{sym} holds given that φ is satisfied. In formal notation, we aim to compute $\Pr[c_{sym} \mid \varphi] = \frac{\Pr[c_{sym} \wedge \varphi]}{\Pr[\varphi]}$.

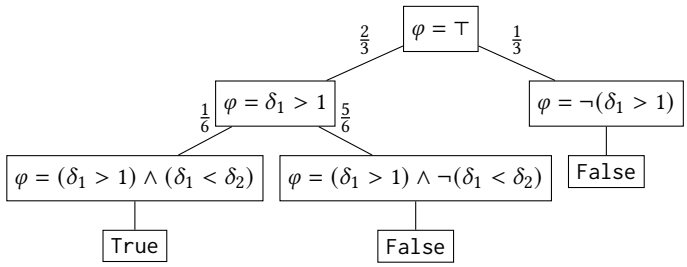
For now we restrict our view to uniform distributions, although we can support weighted distributions without further problems. Note that each probabilistic symbolic variable, δ, is mapped to exactly one distribution, d, and therefore, $\delta \in \text{dom}(d)$. So, assuming there are n probabilistic symbolic variables, $\delta_1, \ldots, \delta_n$, and so n distributions, $d_1, \ldots, d_n$, the set of all possible values $\delta_1, \ldots, \delta_n$ be is $\mathcal{D} = \text{dom}(d_1) \times \cdots \times \text{dom}(d_n)$. We then count the number of elements (or *assignments*) from $\mathcal{D}$ which satisfy $c_{sym} \wedge \varphi$ and φ, and divide these two quantities as shown on line 5 of Alg. 9. Note that p_c is not necessarily a value, but rather a symbolic expression containing constants and universal symbolic variables. Additionally, we exploit 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.

On the *symbolic execution* side, upon reaching the *branch* instruction at Line 25 in Alg 6, the current *state* (S_{cur}) is forked [Line 23] by making two identical copies of it. Furthermore, one additional constraint is appended to each of *newly* created states (S_{true} & S_{false}) encoding the actual *branch condition* (c_{sym}) [Line 28] and it's negation ($\neg c_{sym}$) [Line 29] representing the *true* & *false* side of the *branch*. These states are then added to the *execution* stack containing *states* that need to be explored next. The process continues untill all the *states* in the *execution* stack are explored.

Example. Consider the code in Fig. 5a and suppose we wish to calculate the probability of the program returning `True`. Following Alg. 8 for lines 1,2, 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 $\sigma = \{\texttt{x} \mapsto \delta_1, \texttt{y} \mapsto \delta_2\}$ and $P = \{\delta_1, \delta_2 \mapsto \mathcal{U}\{1,3\}\}$. Now following Alg. 9 to process line 4 of Fig. 5a, note that $\mathcal{D} = \{1,2,3\} \times \{1,2,3\}$ and

```
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



(b) Execution Tree

Fig. 4. An example program and its symbolic execution tree

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

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`.

We use a robust dynamic symbolic execution engine, KLEE to generate the *path constraints* corresponding to each path that our tool explores. For the purpose of our implementation, we modify KLEE to support creation of *probabilistic symbolic* variables whose values can be sampled from a distribution and process the *path* constraints that are stored in the *state* corresponding to the current execution of the program at each of ***assignment*** [Algorithm 6, Line 17], ***branch*** [Algorithm 6, Line 25] & ***sampling*** (`make_pse_symbolic()`) statements. [Algorithm 6, Line 11].

3.3 Query Generation

At this point, we have an execution tree where each branch is annotated with the probability of taking the true and false branch. From this tree we construct a set of paths which are represented by a path condition and set of probabilities. If there are n paths, and φ_i is the i^{th} path condition, and p_{i_j} is the probability of the j^{th} branch in the i^{th} path, then we have $\{(\varphi_1, \{p_{1_1}, \ldots, p_{1_{k_1}}\}), \ldots, (\varphi_n, \{p_{n_1}, \ldots, p_{n_{k_n}}\})\}$. To calculate the probability of an entire path, or equivalently, the probability of a path condition, φ, being true, we can simply multiply each of the branch probabilities. Recall that p_{i_j} is the *conditional* probability of the j^{th} branch condition being true along path i. Since φ_i is the conjunction of each of the k_i branch conditions, the probability of taking the entire path can be computed using the rule $\Pr[A \wedge B] = \Pr[A \mid B]\Pr[B]$ where A and B are any two events. In other words, since

$\varphi_i = c_{i_1} \wedge \cdots \wedge c_{i_{k_i}}$, then

$$\begin{aligned}\Pr[\varphi_i] &= \Pr[c_{i_1} \wedge \cdots \wedge c_{i_{k_i}}] \\ &= \Pr[c_{i_1} \wedge (c_{i_2} \wedge \cdots \wedge c_{i_{k_i-1}} \wedge c_{i_{k_i}})] \\ &= p_{i_1} \cdot \Pr[c_{i_2} \wedge \cdots \wedge c_{i_{k_i}}] \\ &\vdots \\ &= \prod_{j=1}^{k_i} p_{i_j}\end{aligned}$$

So, if we let $p_i = \prod_{j=1}^{k_i} p_{i_j}$, then we can simplify our set of paths to $\Phi = \{(\varphi_1, p_1), \ldots, (\varphi_n, p_n)\}$.

In our interpretation, the ultimate goal of probabilistic symbolic execution is to verify properties of probabilistic programs. In a probabilistic setting this often equates to either proving the upper bound of reaching some "bad" state, or proving the lower bound of reaching some "good" state. We achieve this through queries to an SMT solver, such as Z3 [de Moura and Bjørner 2008]. There are three components to our queries: 1) a universal quantification over the universal symbolic variables, 2) a filtering condition specifying what a "good" or "bad" final state is, and 3) a desired upper/lower bound, potentially parameterized by universal symbolic variables.

(1) If a program has m universal symbolic variables, $\alpha_1, \ldots, \alpha_m$, we begin the query with a universal quantification, $\forall \alpha_1, \ldots, \alpha_m$, in order to reason over any setting of the non-probabilistic program variables.
(2) A *filtering condition*, ψ, is a predicate which determines whether a path is considered "bad" or "good", depending on the property. Some example conditions are whether: a certain value is returned, a false positive (or negative) occurred, or a hash collision occurred. Out of all the paths, Φ, we keep only those which satisfy ψ, Φ'. We then sum over all of the paths in Φ', giving us the probability of ψ occurring in program S:

$$\sum_{(\varphi,p)\in\Phi'} [\varphi] \cdot p.$$

Note that the probability expression p is multiplied by $[\varphi]$, as during some settings of $\alpha_1, \ldots, \alpha_m$, the path represented by φ might not be reachable, and so we should exclude that probability from the sum. The inclusion of the Iverson brackets achieves this desired behavior.
(3) Let δ (???) be the lower/upper bound *expression* which we want to prove the S does not violate.

A general query then takes the form of

$$\forall \alpha_1, \ldots, \alpha_m . \delta \bowtie \sum_{(\varphi,p)\in\Phi'} [\varphi] \cdot p$$

where $\sim$ is a binary relation (e.g. $>, <, \leq, \geq$).

TODO: Finish up once the overview section is done

For example, in the Monty Hall problem as described in Section 2, we were only concerned with those paths that resulted in the contestant winning the car. So, we would filter on $\psi := win = true$, restricting the set of paths to be (TODO: Add on more after writing section 2). The probability of this occurring is then ... as ...

3.4 Formalization

In this section we present the formalization of our method. First, we will present our notation and definitions, and then prove a soundness theorem for our technique. The goal of this section is to describe how $R = (\varphi, \sigma, P)$, the inputs to Alg. 9 is an abstraction of a *distribution of program memories* before a branch guarded by a program expression c.

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

- Let $Vars$ be the set of all program variables, $ForallSymVars$ be the set of all universal symbolic variables, $ProbSymVars$ be the set of all probabilistic symbolic variables, $SymVars = ForallSymVars \cup ProbSymVars$ be the combined set of all symbolic variables, and $Vals$ be the set of all values.
- Let $a_f : ForallSymVars \to Vals$ be an assignment of universal symbolic variables to values and let $ForallAssign$ be the set of all such assignments.
- Similarly, let $a_p : ProbSymVars \to Vals$ be an assignment of probabilistic symbolic variables to values and let $ProbAssign$ be the set of all such assignments.
- Let $m : Vars \to Vals$ be a program memory which translates program variables into values, and let $Mems$ be the set of all program memories.
- Let $de : Mems \to (Vals \to [0, 1])$ be a distribution expression parameterized by program memories, and let $DistExprs$ be the set of all distribution expressions.
- Let $d : ForallAssign \times Mems \to [0, 1]$ be a distribution of program memories parameterized by assignments to universal symbolic variables and let $MemDists$ be the set of all parameterized distributions of program memories.

Additionally, we will use emphatic brackets for two purposes:

- If $e \in ProgExprs$ is a *program* expression containing the program variables $\mathsf{x}_1, \ldots, \mathsf{x}_n \in Vars$, and $m \in Mems$, then

$$[\![e]\!]m = \text{eval}(e[\mathsf{x}_1 \mapsto m(\mathsf{x}_1), \ldots, \mathsf{x}_n \mapsto m(\mathsf{x}_n)])$$

- If $e \in SymExprs$ is a *symbolic* expression containing the symbolic variables $\alpha_1, \ldots, \alpha_n \in ForallSymVars$ and $\delta_1, \ldots, \delta_m \in ProbSymVars$, and $a_f \in ForallAssign$ and $a_p \in ProbAssign$, then

$$[\![e]\!]a_f a_p = \text{eval}(e[\alpha_1 \mapsto a_f(\alpha_1), \ldots, \alpha_n \mapsto a_f(\alpha_n), \delta_1 \mapsto a_p(\delta_1), \ldots, \delta_m \mapsto a_p(\delta_m)])$$

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 $R = (\varphi, \sigma, P)$ be the abstraction generated by the symbolic execution algorithm where $\varphi : ForallAssign \times ProbAssign \to \{0, 1\}$ denotes whether the path condition is true or false under the given assignments, $\sigma : Vars \to SymExprs$ is the mapping from program variables to symbolic expressions generated through symbolic execution, and $P : ForallAssign \to ProbSymVars \rightharpoonup (Vals \to [0, 1])$ is the mapping from probabilistic symbolic variables to the distribution it is sampled from parameterized by assignments of forall symbolic variables. Additionally, for every assignment of forall symbolic variables, $a_f \in ForallAssign$, $\text{domain}(P(a_f)) = \{\delta_1, \ldots, \delta_k\}$. Let $\alpha_1, \ldots, \alpha_l \in ForallSymVars$ be the forall symbolic variables which correspond to the l parameters to the program. For every assignment of probabilistic and forall symbolic variables, $a_f \in ForallAssign, a_p \in ProbAssign$, let $\nu : ForallAssign \to (ProbAssign \to [0, 1])$ be a distribution of assignments of probabilistic symbolic variables parameterized by assignments of forall

symbolic variables, defined as

$$\nu(a_f, a_p) \triangleq \prod_{i=1}^{k} \Pr_{v \sim P(a_f, \delta_i)}[v = a_p(\delta_i)].$$

We say that a distribution d satisfies our abstraction R if, for all assignments of forall symbolic variables, $a_f \in ForallAssign$, $\Pr_{a'_p \sim \nu(a_f)}[\varphi(a_f, a'_p) = 1] > 0$, and if $\mu : ForallAssign \to (ProbAssign \to [0, 1])$ is defined as

$$\mu(a_f, a_p) = \frac{\Pr_{a'_p \sim \nu(a_f)}[a'_p = a_p \wedge \varphi(a_f, a'_p) = 1]}{\Pr_{a'_p \sim \nu(a_f)}[\varphi(a_f, a'_p) = 1]}.$$

Additionally, define convertToMem : $(Vars \to SymExprs) \to ForallAssign \to ProbAssign \to Mem$ as

$$\text{convertToMem}(\sigma, a_f, a_p) \triangleq \lambda(\mathsf{x} : Vars)\ .\ [\![\sigma(\mathsf{x})]\!] a_f a_p,$$

and let $\text{convertFromMem}(\sigma, a_f, m) = (\text{convertToMem}(\sigma, a_f))^{-1}(m)$. Then,

$$d(a_f, m) = \sum_{a_p \in \text{convertFromMem}(\sigma, a_f, m)} \mu(a_f, a_p).$$

We additionally define the semantics for the three main types of statements which concerns probabilistic symbolic execution: assignments, probabilistic samples, and branches.

Definition 3.2 (Assignment Semantics). Let $d \in MemDists$ be a distribution of program memories parameterized by assignments to forall symbolic variables. Let $\mathsf{x} = e$ be an arbitrary assignment of the program variable $\mathsf{x} \in Vars$ to the program expression $e \in ProgExprs$. Let $\text{assign}_{\mathsf{x}=e} : Mems \to Mems$ is defined as

$$\text{assign}_{\mathsf{x}=e}(m) = \lambda(\mathsf{y} : Vars) \begin{cases} [\![e]\!] m & \text{if } \mathsf{x} = \mathsf{y} \\ m(\mathsf{y}) & \text{otherwise} \end{cases}$$

and let $\text{unassign}_{\mathsf{x}=e} = \text{assign}^{-1}_{\mathsf{x}=e}$. Then we define $d_{\mathsf{x}=e}$ to be the distribution over program memories parameterized by assignments to forall symbolic variables after executing the assignment statement $\mathsf{x} = e$ to be

$$d_{\mathsf{x}=e}(a_f, m) = \sum_{m' \in \text{unassign}_{\mathsf{x}=e}(m)} d(a_f, m').$$

Definition 3.3 (Sampling Semantics). Let $d \in MemDists$ be a distribution of program memories parameterized by assignments to forall symbolic variables. Let $\mathsf{x} \sim de$ be an arbitrary sampling instruction which assigns the program variable $\mathsf{x} \in Vars$ to a random element from the distribution of values parameterized by a memory, represented as a distribution expression $de \in DistExprs$. Let $\text{desample} : Vars \times Mems \to \mathcal{P}(Mems)$ be defined as

$$\text{desample}(\mathsf{x}, m) = \{m' \in Mems \mid \forall(\mathsf{y} \in Vars)\ .\ (\mathsf{y} \neq \mathsf{x} \wedge m'(\mathsf{y}) = m(\mathsf{y}))\}.$$

Then, we define $d_{\mathsf{x} \sim de}$ to be the distribution over program memories parameterized by assignments to forall symbolic variables after executing the sampling statement $\mathsf{x} \sim de$ to be

$$d_{\mathsf{x} \sim de}(a_f, m) = \sum_{m' \in \text{desample}(m)} ([\![de]\!] a_f)(m(\mathsf{x})) \cdot d(a_f, m').$$

Definition 3.4 (Conditional Distribution of Program Memories). Let $a_f \in ForallAssign$ be an arbitrary assignment of forall symbolic variables, c be a guard of an `if` condition, $d \in MemDists$ be a distribution of program memories parameterized by assignments to forall symbolic variables, and

$x_1, \ldots, x_n \in Vars$ be all of the program variables in c. Then for all program memories $m \in Mems$, d conditioned on a guard c being true, represented as d_c is defined as

$$d_c(a_f, m) = \frac{\Pr_{m' \sim d(a_f)}\ [m' = m \wedge [\![c]\!]m' = \texttt{true}]}{\Pr_{m' \sim d(a_f)}\ [[\![c]\!]m' = \texttt{true}]}$$

Similarly, for all program memories $m \in Mems$, d conditioned on a guard c being false, represented as $d_{\neg c}$ is defined as

$$d_{\neg c}(a_f, m) = \frac{\Pr_{m' \sim d(a_f)}\ [m' = m \wedge [\![c]\!]m' = \texttt{false}]}{\Pr_{m' \sim d(a_f)}\ [[\![c]\!]m' = \texttt{false}]}$$

3.4.2 Proofs.

4 IMPLEMENTATION

KLEE Description Moved to §3.1
KLEE Modifications Moved to §3.2

5 CASE STUDIES

In this section we will briefly explain each of the case studies that we use in our evaluation (Section 6). For each case study, we will explain, (1) what the algorithm does, (2) which variables are concretized, universally quantified, and probabilistic, and (3) the property we aim to verify using our technique. Note that we frame the queries as an existential query and hope to get "UNSAT" in order to reason over all the possible values for the universal symbolic variables.

5.1 Freivalds' Algorithm

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 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 outputs "Yes", and "No" otherwise. However, if $A \times B \neq C$, the probability that the algorithm returns "Yes" is at most $\frac{1}{2}$.

While the size of the matrices has to be concretized, the elements of the three matrices can be represented by universal symbolic variables and the elements of r as probabilistic symbolic variables. We want to verify the false positive error rate of $\frac{1}{2}$. To do this, we can ask Z3, for a fixed n, whether there exist any $n \times n$ matrices A, B, and C where $A \times B \neq C$ such that

$$\Pr[\texttt{freivalds}(A, B, C) = \text{Yes}] > \frac{1}{2}.$$

5.2 Randomized Response

Randomized response is a surveying technique which allows respondents to answer in a way that provides "plausible deniability" Before answering the query, a coin is flipped. If "tails", then the respondent answers truthfully, if "heads", a second coin is flipped and the respondent answers "Yes" if "heads" and "No" if tails. In fact, this method is $(\ln 3, 0)$-differentially private.

The answer the respondent would give if they answered truthfully can be represented with a universal symbolic variable, and the results of the two coin flips as probabilistic symbolic variables. With this model, we can prove $(\ln 3, 0)$ differential privacy by asking our method if there exists a setting of the "truth" such that

$$\frac{\Pr[\text{Response} = \text{Yes} \mid \texttt{text} = \text{Yes}]}{\Pr[\text{Response} = \text{Yes} \mid \texttt{text} = \text{No}]} \neq 3$$

and

$$\frac{\Pr[\text{Response} = \text{No} \mid \texttt{text} = \text{No}]}{\Pr[\text{Response} = \text{No} \mid \texttt{text} = \text{Yes}]} \neq 3.$$

5.3 Reservoir Sampling

Algorithm 5 Reservoir Sampling

```
1: function ReservoirSampling(A[1..n], S[1..k])
2:     for i = 1 to k do
3:         S[i] ← A[i]
4:     for i = k + 1 to n do
5:         j ← UniformInt(1, i)
6:         if j <= k then
7:             S[j] ← A[i]
8:     return S
```

Reservoir sampling 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 maintain a set of k elements drawn from the set of n elements. For the full algorithm, see Alg. 5.

The sizes of both the population, n, and the sample, k, need to be concretized. The elements from A, are universal symbolic variables, and each sample, j, are probabilistic symbolic variables. The property that we want to check is whether each sample has an equal probability of being returned by Alg. 5, namely $\frac{1}{\binom{n}{k}}$.

5.4 Schwartz-Zippel Lemma

The Schwartz-Zippel lemma is a probabilistic method of polynomial identity testing, that is, the problem of determining whether a given multivariate polynomial is identially equal to 0 or not. Given a non-zero polynomial of total degree d, P, over a field F, and $r_1, \ldots, r_n$ selected at random from a subset of F, say S, then the lemma states that

$$\Pr[P(r_1, \ldots, r_n) = 0] \leq \frac{d}{|S|}.$$

The number of terms in P, n, has to be concretized, the degrees of each of the terms, and the size of the subset S all need to be concretized. The coefficients are represented as universal symbolic variables and $r_1, \ldots, r_n$ are probabilistic symbolic variables. The property that we want to check is whether there exists a non-zero polynomial, P, such that

$$\Pr[P(r_1, \ldots, r_n) = 0] > \frac{d}{|S|}.$$

5.5 Bloom Filter

A Bloom filter is a space-efficient, probabilistic data structure used to rapidly determine whether an element is in a set. A Bloom filter 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.

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

$$n = -\frac{m \ln \varepsilon}{(\ln 2)^2}$$

$$k = -\frac{\ln \varepsilon}{\ln 2}$$

We want to prove that for a given m and ε that the actual false positive rate does not exceed ε.

With our method, m and ε first need to be concretized. Then, we insert m elements x_i, where each x_i can be represented using a universal symbolic variable. We model each of the k uniform hash functions using the method described in Sec. (TODO: Insert bit about hash functions). We then want to check if there exist $x_1, \ldots, x_m$ such that the false positive rate exceeds ε.

5.6 Quicksort

Quicksort is a popular sorting algorithm which uses partitioning in order to achieve efficient sorting. One way to choose a pivot element is by way of a uniform random sample. Using this pivot method, the expected number of pivots required is $1.386n \log_2(n)$ where n is the length of the array. If we concretize the length of the array, but represent the elements as universal symbolic variables, we can compute the expected number of pivots required to sort the array.

6 EVALUATION

7 RELATED WORK

Subhajit & Sumit: If you could both start looking into related work (Mayhap, original PSE paper, Axprof, P4WN, PSI), that would be great!

Write a paragraph or two about use of symbolic execution from other papers. In what ways has symbolic execution has been used and some traditional aspects. Write about AxProf, Original PSE paper [ISSTA 2012], Dice [Scaling Exact Inference for Discrete Probabilistic Programs, OOPSLA'20], Quantification of Software Changes through Probabilistic Symbolic Execution [ASE 2015], P4WN [ASPLOS'21], PSI [PLDI'20], Exact Inference for Higher-order Probabilistic Programs paper [FSE'15 QCoral] etc. Write the central idea in these papers and contrast with differences from approach presented in this paper.

8 CONCLUSION & FUTURE WORK

ACKNOWLEDGMENTS

REFERENCES

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.

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.

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

A APPENDIX

Text of appendix ...

B WIP PROBABILISTIC SYMBOLIC EXECUTION

In this section, we present our technique for augmenting traditional symbolic execution to support probabilistic programs with discrete sampling instructions. We begin with a short review of a traditional symbolic execution algorithm and then discuss how we calculate exact path probabilities.

Symbolic execution is a program analysis technique where a program is run on *symbolic* inputs and all program operations are replaced with those which manipulate these symbolic variables. We present a high-level description of how *symbolic execution* works in Alg 6. For each path in the program, a list of *constraints* are stored that encodes the whole path as a *logical* formula. These constraints are over *program* variables that have been marked *symbolic* by the user and are stored in the *state* data structure as *metadata* during the *dynamic* symbolic execution of the program. Apart from *path constraints*, the *state* also contains a mapping of *symbolic* variables to expressions that are either *symbolic* or *concrete* (in the case of concrete execution of the program) and a list of *instructions* currently getting executed as a part of the *state*. During symbolic execution, program state is encoded symbolically in two parts: a conjunctive formula, φ, consisting of the branch conditions which are true for that particular path, called a *path condition*, and a mapping from program variables to symbolic expressions containing constants and *symbolic variables*, σ. Each *instruction* in the program is executed and depending upon the *type* of the *instruction* the *state* along with the *symbolic variables* mapping, (σ in Alg 6) are updated as described in Algorithm 6. *Symbolic variables* are assigned values by employing an SMT Solver for solving the current set of *constraints* collected so far in the *program path* and also by *concretizing* some of the values that get evaluated as a result of *concrete* execution (e.g. return value from external function calls). When execution reaches an assignment of the form $\mathsf{x} = e$, where e is a constant or program variable, $\sigma[\mathsf{x}]$ is either just the constant, e, or the symbolic representation of e, namely $\sigma[e]$. If e is an expression, we recursively convert each program variable in e to its corresponding symbolic expression found in σ to construct a new symbolic execution, e_{sym}, and set $\sigma[\mathsf{x}] = e_{sym}$. When execution reaches a branch guarded by the condition c, execution proceeds down both branches, one where $\varphi = \varphi \wedge \sigma[c]$, and the other where $\varphi = \varphi \wedge \neg\sigma[c]$.

B.1 Adding Probabilistic Sampling

We present the formalism of our approach on **pWhile**, a core imperative probabilistic programming language, as a model for more general languages:

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

Above, x is a program variable, e is an expression, and d is a *discrete* distribution expression which denotes which distribution the sample should be drawn from. We permit distribution expressions to be (optionally) parameterized by program variables. For example, $\mathsf{UniformInt}(1, \mathsf{x})$ is a uniform distribution which selects at random a value between 1 and x (inclusive), where $\mathsf{x} \geq 1$. In order to support sampling instructions we make the following additions to traditional symbolic execution:

- *Probabilistic symbolic variables.* We distinguish two types of symbolic variables: *universal* symbolic variables (identical to those in traditional symbolic execution), and *probabilistic* symbolic variables. For each sampling instruction a new probabilistic symbolic variable is created to denote the result of sample.
- *Distribution map.* We add a new mapping from probabilistic symbolic variables to distribution expressions, P, which 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.

Algorithm 6 Probabilistic Symbolic Execution Algorithm

```
1:  function SymbEx(Prog : Program)
2:      φ_paths ← [], L ← [], Enc_t ← ⊥, Enc_a ← ⊥                ▷ Initialization
3:      I_0 ← getStartInstruction(Prog)
4:      S_0 ← (I_0, φ, φ, φ, 1.0)                                  ▷ Empty Initial State
5:      L.Append(S_0)                                              ▷ Start with S_cur in Execution Stack
6:      while L ≠ φ do
7:          (I_cur, φ, σ, P, p) ← L.Remove( )
8:          if UNSAT(φ) ∨p = 0.0 then
9:              continue
10:         switch instType(I_cur) do
11:             case x ← d                                         ▷ Create fresh Symbolic Variable
12:                 I_1 ← getNextInstruction(I_cur)
13:                 P_1 ← PSESample(x, d, P)
14:                 S_1 ← (I_cur, φ, σ, P_1, p)
15:                 L.Append(S_1)
16:             end case
17:             case v ← e                                         ▷ Assignment Instruction
18:                 I_1 ← getNextInstruction(I_cur)
19:                 σ[x] ← σ[e]
20:                 S_cur ← (I_cur, φ, σ, P)
21:                 L.Append(S_cur)
22:             end case
23:             case if (c) then goto T_1                          ▷ Branch Instruction
24:                 c_sym ← σ[c]
25:                 p_t, p_f ← PSEBranch(c_sym, σ, P)
26:                 I_1 ← getInstruction(T_1)
27:                 I_2 ← getNextInstruction(I)
28:                 S_true ← (I_1, φ ∧ c_sym, σ, p_t)
29:                 S_false ← (I_2, φ ∧ ¬c_sym, σ, p_f)
30:                 L.Append(S_false)
31:                 L.Append(S_true)                               ▷ Start with True State
32:             end case
33:             case halt(c)                                       ▷ Terminate Instruction
34:                 c_sym ← σ[c]
35:                 if SAT(c_sym) then
36:                     Enc_t.AddEncoding(φ, p)
37:                 Enc_a.AddEncoding(φ, p)
38:             end case
39:         end switch
40:     r ← Solve(Enc_t, Enc_a)
41: return r
```

- *Path probability.* For each path, we adjoin a path probability expression, p, which is parameterized by universal symbolic variables. Now each path can be identified by both its path condition and its probability.

Algorithm 7 PSE Assignment Algorithm

1: **function** PSEAssignment(x, e, φ, σ, P)
2: $e_{sym} \leftarrow \sigma[e]$
3: $\sigma[\mathsf{x}] = e_{sym}$
4: **return** (φ, σ, P)

Assignment. For assignment statements of the form $\mathsf{x} \leftarrow e$, where x is a *program* variable and e is an expression, probabilistic symbolic execution proceeds identically to traditional symbolic execution, as detailed in Alg. 7. On the *symbolic execution* side, the *symbolic variable* mapping (σ) is updated with the result of the assignment operation is stored back in the current *state* for further execution as shown in Line 17 of Alg 6.

Algorithm 8 PSE Sampling Algorithm

1: **function** PSESample(x, d, φ, σ, P)
2: $\delta \leftarrow$ Generate a fresh probabilistic symbolic variable
3: $\sigma[\mathsf{x}] = \delta$
4: $P[\delta] = d$
5: **return** (φ, σ, P)

Sampling. For sampling statements, $\mathsf{x} \overset{\$}{\leftarrow} d$, where x is a *program* variable and d is a distribution expression, Alg. 8 is used. A fresh probabilistic symbolic variable, δ, is created, σ is updated to be $\sigma[\mathsf{x}] = \delta$, and the original distribution d is recorded in P by setting $P[\delta] = d$. On the *symbolic execution* side, the updated *Distribution Map* (P) with the result of sampling operation is stored back in the current *state* for further execution as shown in Line 11 of Alg 6.

Algorithm 9 PSE Branch Algorithm

1: **function** PSEBranch(c, φ, σ, P)
2: $(\delta_1, \ldots, \delta_n) \leftarrow \mathrm{dom}\,(P)$
3: $(d_1, \ldots, d_n) \leftarrow (P[\delta_1], \ldots, P[\delta_n])$
4: $p_c \leftarrow \dfrac{\sum_{(v_1,\ldots,v_n) \in \mathrm{dom}(d_1)\times\cdots\times\mathrm{dom}(d_n)} [(c_{sym} \wedge \varphi)\{\delta_1 \mapsto v_1, \ldots, \delta_n \mapsto v_n\}]}{\sum_{(v_1,\ldots,v_n) \in \mathrm{dom}(d_1)\times\cdots\times\mathrm{dom}(d_n)} [\varphi\{\delta_1 \mapsto v_1, \ldots, \delta_n \mapsto v_n\}]}$
5: **return** $((\varphi \wedge c_{sym}, \sigma, P, p_c), (\varphi \wedge \neg c_{sym}, \sigma, P, 1 - p_c))$

Branches (Alg. 9). Note that with the inclusion of probabilistic symbolic variables we can now either branch on universal or probabilistic symbolic variables (or both). Intuitively, branches are handled much the same in probabilistic symbolic execution as they are in traditional symbolic execution save for one detail: now branches have a probability associated with them. Given a guard expression c, how do we compute the probability of c being true?

To gain intuition, for now just consider the special case where the guard condition only references universal symbolic variables. Let c_{sym} be the equivalent symbolic expression for the guard c and assume that c_{sym} does not reference any probabilistic symbolic variables. Note that which side of the branch is taken is solely determined by the setting of the univeral symbolic variables. Therefore,

one side of the branch must have a probability of 1, and the other side, 0. 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 $[\neg c_{sym}]$.

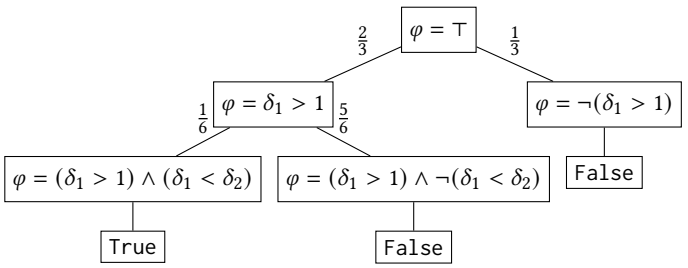
For probabilistic branches, i.e. guards which branch on probabilistic symbolic variables, computing the branch probability is trickier as given a fixed setting of the universal symbolic variables, it is unclear which branch execution will follow as this is dependent upon the sampling results. Without loss of generality, consider a branch of the form **if** c **then** S_1 **else** S_2. As before, we define $c_{sym} = \sigma[c]$, or the symbolic expression representation of the guard expression, c, and we want to compute the probability of taking the "true" branch and the "false" branch, assuming execution has reached the start of the `if` condition. Since the path condition φ records the necessary constraints on the universal and probabilistic symbolic variables which must hold in order to reach this `if` condition, we can view this probability a *conditional probability*, or the probability that c_{sym} holds given that φ is satisfied. In formal notation, we aim to compute $\Pr[c_{sym} \mid \varphi] = \frac{\Pr[c_{sym} \wedge \varphi]}{\Pr[\varphi]}$.

For now we restrict our view to uniform distributions, although we can support weighted distributions without further problems. Note that each probabilistic symbolic variable, δ, is mapped to exactly one distribution, d, and therefore, $\delta \in \text{dom}\,(d)$. 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 $\delta_1, \ldots, \delta_n$ be is $\mathcal{D} = \text{dom}\,(d_1) \times \cdots \times \text{dom}\,(d_n)$. We then count the number of elements (or *assignments*) from $\mathcal{D}$ which satisfy $c_{sym} \wedge \varphi$ and φ, and divide these two quantities as shown on line 5 of Alg. 9. Note that p_c is not necessarily a value, but rather a symbolic expression containing constants and universal symbolic variables. Additionally, we exploit 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.

On the *symbolic execution* side, upon reaching the *branch* instruction at Line 25 in Alg 6, two new *states* are created (S_{true} & S_{false}) and the constraint encoding the actual *branch condition* (c_{sym}) [Line 28] and it's negation ($\neg c_{sym}$) [Line 29] are appended to them representing the *true* & *false* side of the *branch*. These states are then added to the *execution* stack containing *states* that need to be explored next. The process continues untill all the *states* in the *execution* stack are explored.

```
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



(b) Execution Tree

Fig. 5. An example program and its symbolic execution tree

Example. Consider the code in Fig. 5a and suppose we wish to calculate the probability of the program returning `True`. Following Alg. 8 for lines 1,2, 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 $\sigma = \{\texttt{x} \mapsto \delta_1, \texttt{y} \mapsto \delta_2\}$ and $P = \{\delta_1, \delta_2 \mapsto \mathcal{U}\{1, 3\}\}$. Now following Alg. 9 to process line 4 of Fig. 5a, 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`.

We use a robust dynamic symbolic execution engine, KLEE to generate the *path constraints* corresponding to each path that our tool explores. For the purpose of our implementation, we modify KLEE to support creation of *probabilistic symbolic* variables whose values can be sampled from a distribution and process the *path* constraints that are stored in the *state* corresponding to the current execution of the program at each of ***assignment*** [Algorithm 6, Line 17], ***branch*** [Algorithm 6, Line 25] & ***sampling*** (`make_pse_symbolic()`) statements. [Algorithm 6, Line 11].