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), 12 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 [**?**] 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.

We can represent this problem as a probabilistic program, as shown in Fig. 1, where `choice` $\in [1, 3]$ and 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. Supposedly 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. The problem thus becomes, given a probabilistic program with discrete sampling statements, and a program property, how do we verify that the program satisfies the property?

One typical solution would be to run a simulation on random inputs and compute the probability of winning if `door_switch` = `true`, and the probability of winning if `door_switch` = `false`. While

*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/

Fig. 1. C code for the Monty Hall Problem

for this program this would work as the input space is tiny, the resulting output distributions would be absent any formal guarantees and only be an approximation.

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

Subhajit & Sumit: If you could add a high-level description of symb. exec., maybe an algorithm, that would be great!

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. During execution, program state is encoded symbolically in two parts: a path condition which is a conjunctive formula, φ, which records the branch conditions which are true for that particular path, and a mapping from program variables to symbolic expressions containing constants and *symbolic variables*, σ. When execution reaches an assignment of the form $\mathsf{x} = e$, where e is a constant or program variable, $\sigma[\mathsf{x}] = \sigma[e]$. Similarly, if $e = e_1 \oplus e_2$, where $\oplus$ is an arbitrary binary operation, $\sigma[\mathsf{x}] = \sigma[e_1] \oplus \sigma[e_2]$. 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 now consider the problem of performing symbolic execution on a simple imperative probabilistic programming language, **pWhile**:

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

Above, x is a program variable, e is an expression, and d is a *discrete* distribution expression. In order to support sampling instructions we make the following additions to traditional symbolic execution:

- *Probabilistic symbolic variables.* We distiguish two types of symbolic variables: *universal* symbolic variables (identical to those in traditional symbolic execution), and *probabilistic* symbolic variables. For each unique 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.
- *Path probability.* For each path, we adjoin a path probability expression, p, which is parameterized by universal symbolic variables.

Algorithm 1 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)
5: **end function**

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. 1.

Algorithm 2 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)
6: **end function**

Sampling. For sampling statements, $\mathsf{x} \xleftarrow{\$} d$, where x is a *program* variable and d is a distribution expression, Alg. 2 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 $P[\delta] = d$.

Algorithm 3 PSE Branch Algorithm

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

Branches. Note that with the inclusion of probabilistic symbolic variables we can now either branch on universal or probabilistic symbolic variables (or both). For guards whose symbolic version does not contain any probabilistic symbolic variables, probabilstic symbolic execution works nearly identically to traditional symbolic execution, save one detail: the probability of taking the branch. If c_{sym} is a symbolic expression which represents the guard to a branch, and c_{sym} does not contain any probabilistic symbolic variables, then we define the probability of taking the "true" branch is $[c_{sym}]$, and the probability of taking the "false" branch is $[\neg c_{sym}]$, where $[Q] = 1$ if Q is true, and 0 otherwise (known as Iverson brackets).

For probabilistic branches, i.e. guards which branch on probabilistic symbolic variables, Alg. 3 is used instead of the traditional symbolic execution branch algorithm. Without loss of generality,

Algorithm 4 Getting Path Constraints : Symbolic Execution

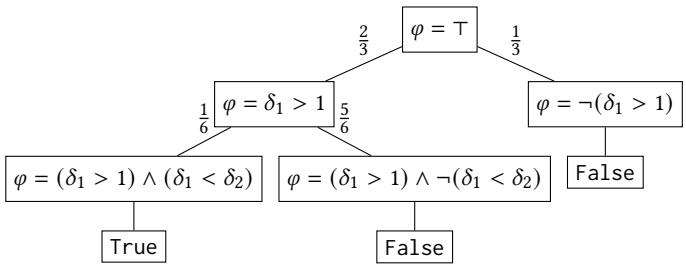
```
1:  function SymbEx(P_prog : Program)
2:      φ_paths ← [], Ex_stack ← [], Δ ← [], ϕ_paths ← ϕ                ▷ Initialization
3:      I_0 ← getStartInstruction(P_prog)
4:      S_0 ← [I_0, ϕ_path, Δ]                                          ▷ Empty Initial State
5:      Ex_stack.Push(S_0)                                              ▷ Start with S_cur in Execution Stack
6:      while Ex_stack != ϕ do
7:          S_cur ← Ex_stack.Pop( ), I_cur ← S_cur[1]                   ▷ Start State, [I_0, ϕ_path, Δ]
8:          switch instType(I_cur) do
9:              case v := e                                             ▷ Assignment Instruction
10:                 I_cur ← getNextInstruction(I_cur)
11:                 S_cur ← [I_cur, ϕ_path, Δ[v → Eval(e, Δ)]]
12:                 Ex_stack.Push(S_cur)
13:             case if (c_sym) then P_1 else P_2                       ▷ Branch Instruction
14:                 if (isSat(c_sym) ∧ isSat(¬c_sym)) then              ▷ Both cases SAT
15:                     I_1 ← getStartInstruction(P_1)
16:                     I_2 ← getStartInstruction(P_2)
17:                     S_true ← [I_1, ϕ_path ∧ c_sym, Δ]
18:                     S_false ← [I_2, ϕ_path ∧ (¬c_sym), Δ]
19:                     Ex_stack.Push(S_false)
20:                     Ex_stack.Push(S_true)                           ▷ Start with True State
21:                 else if isSat(c_sym) then                           ▷ True case SAT
22:                     I_cur ← getStartInstruction(P_1)
23:                     S_cur ← [I_cur, ϕ_path ∧ c_sym, Δ]
24:                     Ex_stack.Push(S_cur)
25:                 else if isSat(¬c_sym) then                          ▷ False case SAT
26:                     I_cur ← getStartInstruction(P_2)
27:                     S_cur ← [I_2, ϕ_path ∧ (¬c_sym), Δ]
28:                     Ex_stack.Push(S_cur)
29:                 else                                                ▷ Unconditional Branch
30:                     I_cur ← getNextInstruction(I_cur)
31:                     S_cur ← [I_cur, ϕ_path, Δ]
32:                     Ex_stack.Push(S_cur)
33:                 end if
34:             case HALT                                               ▷ Terminate Instruction
35:                 φ_paths ← φ_paths.Append(ϕ_path)
36:     end while
37: return φ_paths
38: end function
```

consider a branch of the form **if** c **then** S_1 **else** S_2 where c is a probabilistic branch. As before, we define $c_{sym} = \sigma[c]$. The core of Alg. 3 is computing the probability of taking each branch. We interpret probabilistic branches as a conditioning operation on the distributions which are mentioned in the guard of the branch. Under this interpretation, we aim to compute the conditional probability $\Pr[c_{sym} \mid \varphi] = \frac{\Pr[c_{sym} \wedge \varphi]}{\Pr[\varphi]}$, where φ is the current path condition formula.

We use a form of model counting in order to compute p_c, the probability of c being true. 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. 3. 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.

```
1: x ←$ UniformInt(1, 3)
2: y ←$ UniformInt(1, 3)
3: if x > 1 then
4:     if x < y then
5:         return True
6:     end if
7: else
8:     return False
9: end if
```

(a) Program



(b) Execution Tree

Fig. 2. An example program and its symbolic execution tree

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

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

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 multiple 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

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

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 [**?**]. 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" 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 occuring 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

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

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 occuring 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 provide proofs of correctness and soundness of our technique.

3.4.1 Notation & Definitions. The goal of this section is to describe how $R = (\varphi, \sigma, P)$, the inputs to Alg. 3 is an abstraction of a *distribution of program memories* before a branch guarded by a program expression c. 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,

$a_f \in \mathit{ForallAssign}, a_p \in \mathit{ProbAssign}$, let $\nu : \mathit{ForallAssign} \to (\mathit{ProbAssign} \to [0,1])$ 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, $a_f \in \mathit{ForallAssign}$, $\Pr_{a'_p \sim \nu(a_f)}[\varphi(a_f, a'_p) = 1] > 0$, and if $\mu : \mathit{ForallAssign} \to (\mathit{ProbAssign} \to [0,1])$ 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 $\mathrm{convertToMem} : (\mathit{Vars} \to \mathit{SymExprs}) \to \mathit{ForallAssign} \to \mathit{ProbAssign} \to \mathit{Mem}$ as

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

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

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

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 $d \in \mathit{MemDists}$ 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 $\mathsf{x} \in \mathit{Vars}$ to the program expression $e \in \mathit{ProgExprs}$. Let $\mathrm{assign}_{\mathsf{x}=e} : \mathit{Mems} \to \mathit{Mems}$ is defined as

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

and let $\mathrm{unassign}_{\mathsf{x}=e} = \mathrm{assign}^{-1}_{\mathsf{x}=e}$. 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

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

Definition 3.3 (Sampling Semantics). Let $d \in \mathit{MemDists}$ 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 $\mathsf{x} \in \mathit{Vars}$ to a random element from the distribution of values parameterized by a memory, represented as a distribution expression $de \in \mathit{DistExprs}$. Let $\mathrm{desample} : \mathit{Vars} \times \mathit{Mems} \to \mathcal{P}(\mathit{Mems})$ be defined as

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

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

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

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 $\mathtt{x}_1, \ldots, \mathtt{x}_n \in Vars$ 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

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

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

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

3.4.2 *Proofs.*

4 IMPLEMENTATION

Subhajit & Sumit: Write something about KLEE and the implementation here!

We use a robust dynamic symbolic execution engine, KLEE to generate the *path constraints* corresponding to each path that our tool explores. For each path in the program, KLEE stores a list of *constraints* that encodes the whole path. These constraints are over *program* variables that have been marked `make_symbolic()` by the user and are stored in KLEE as *metadata* in the *state* data structure 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 *dynamic* execution, KLEE executes each *instruction* in the program and updates the *state* along with the *symbolic* variables mapping (Δ) depending upon *type* of the *instruction* it executes as described in Algorithm 4. KLEE assigns values to *symbolic* variables by employing an SMT Solver (Z3 as in our case) for solving the current set of *constraints* seen so far in the program path and also by concretizing some of the values that get evaluated as a result of *concrete* execution (Eg. return value from external function calls).

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 dump the whole set of *path* constraints that is stored in the *state* corresponding to the current execution of the program at each of (1) ***assignment*** [Algorithm 4, Line 9], (2) ***branch*** [Algorithm 4, Line 13], & (3) ***assume*** [Algorithm 4, Line **??**] statements. These dumps are later refined and used by the tool for further processing.

Upon reaching the *branch* instruction at Line 13 in Algorithm 4, based on whether the *path constraints* are solvable upon adding the *branch* [Line 13] condition, KLEE forks the current *state* by making two identical copies of it and then appending to each of *new* states one additional constraint encoding the *true* [Line 17] and *false* [Line 18] side of the branch and adds it 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.

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 [**?**] 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:      end for
5:      for i = k + 1 to n do
6:          j ← UniformInt(1, i)
7:          if j <= k then
8:              S[j] ← A[i]
9:          end if
10:     end for
11:     return S
12: end function
```

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!

8 CONCLUSION & FUTURE WORK

ACKNOWLEDGMENTS

A APPENDIX

Text of appendix …