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), 7 pages.

1 INTRODUCTION

1.1 Contributions

2 MOTIVATING EXAMPLE

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 $\varphi = \varphi \wedge \neg\sigma[c]$.

[*] 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/

3.2 Adding Probabilistic Sampling

We now consider the problem of performing symbolic execution on a simple imperative probabilistic programming language, **pWhile**:

$$P := \textbf{skip} \mid \mathsf{x} \leftarrow e \mid \mathsf{x} \overset{\$}{\leftarrow} d \mid P; P \mid \textbf{if } e \textbf{ then } P \textbf{ else } P \mid \textbf{while } e \textbf{ do } P$$

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: *universally quantified* 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 universally quantified symbolic variables.

Algorithm 1 PSE Assignment Algorithm

```
function PSEAssignment($\mathsf{x}, e, \varphi, \sigma, P, I$)
    $e_{sym} \leftarrow$ Convert $v$ into a symbolic expression using $\sigma$
    $\sigma[\mathsf{x}] = e_{sym}$
    return $(\varphi, \sigma, P, I)$
end function
```

Algorithm 2 PSE Sampling Algorithm

```
function PSESample($\mathsf{x}, d, \varphi, \sigma, P$)
    $\delta \leftarrow$ Generate a fresh probabilistic symbolic variable
    $\sigma[\mathsf{x}] = \delta$
    $P[\delta] = d$
    return $(\varphi, \sigma, P)$
end function
```

When a sampling statement, $\mathsf{x} \overset{\$}{\leftarrow} d$, is reached, 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 $P[\delta] = d$. Note that with the inclusion of probabilistic symbolic variables we can now either branch on universally quantified 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 $[\cdot]$ are 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 3 PSE Branch Algorithm

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

consider a branch of the form **if** c **then** P_1 **else** P_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 $\Pr[c_{sym} \mid \varphi]$ 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, $\delta \in \mathrm{dom}\,(d)$. 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 $\mathcal{D} = \mathrm{dom}\,(d_1) \times \cdots \times \mathrm{dom}\,(d_n)$. We then count the number of *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. In Section 3.3 we have a proof of correctness for this quantity, but the intuition is by applying the definition of conditional probability. Note that p_c is not necessarily a value, but rather a symbolic expression containing constants and universally quantified symbolic variables. 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.

3.3 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.

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

Algorithm 4 Getting Path Constraints : Symbolic Execution

```
function SymbEx($P_{prog}$ : $Program$)
    $\varphi_{paths} \leftarrow []$, $Ex_{stack} \leftarrow []$        ▷ List to store all the path conditions.
    $I_0 \leftarrow$ getStartInstruction($P_{prog}$)
    $S_0 \leftarrow [I_0, \phi_{path}, \Delta]$        ▷ Empty Initial State
    $Ex_{stack}$.Push($S_0$)        ▷ Start with $S_{cur}$ in Execution Stack
    while $Ex_{stack}$ != $\phi$ do
        $S_{cur} \leftarrow Ex_{stack}$.Pop( ), $I_{cur} \leftarrow S_{cur}[1]$        ▷ Start State, $[I_0, \phi_{path}, \Delta]$
        switch instType($I_{cur}$) do
            case v := e        ▷ Assignment Instruction
                $I_{cur} \leftarrow$ getNextInstruction($I_{cur}$)
                $S_{cur} \leftarrow [I_{cur}, \phi_{path}, \Delta[\text{v} \rightarrow \text{Eval}(e, \Delta)]]$
                $Ex_{stack}$.Push($S_{cur}$)
            case if ($c_{sym}$) then $P_1$ else $P_2$        ▷ Branch Instruction
                if (isSat($c_{sym}$) $\wedge$ isSat($\neg c_{sym}$)) then        ▷ Both cases SAT
                    $I_1 \leftarrow$ getStartInstruction($P_1$)
                    $I_2 \leftarrow$ getStartInstruction($P_2$)
                    $S_{true} \leftarrow [I_1, \phi_{path} \wedge c_{sym}, \Delta]$
                    $S_{false} \leftarrow [I_2, \phi_{path} \wedge (\neg c_{sym}), \Delta]$
                    $Ex_{stack}$.Push($S_{false}$)
                    $Ex_{stack}$.Push($S_{true}$)        ▷ Start with True State
                else if isSat($c_{sym}$) then        ▷ True case SAT
                    $I_{cur} \leftarrow$ getStartInstruction($P_1$)
                    $S_{cur} \leftarrow [I_{cur}, \phi_{path} \wedge c_{sym}, \Delta]$
                    $Ex_{stack}$.Push($S_{cur}$)
                else if isSat($\neg c_{sym}$) then        ▷ False case SAT
                    $I_{cur} \leftarrow$ getStartInstruction($P_2$)
                    $S_{cur} \leftarrow [I_2, \phi_{path} \wedge (\neg c_{sym}), \Delta]$
                    $Ex_{stack}$.Push($S_{cur}$)
                else        ▷ Unconditional Branch
                    $I_{cur} \leftarrow$ getNextInstruction($I_{cur}$)
                    $S_{cur} \leftarrow [I_{cur}, \phi_{path}, \Delta]$
                    $Ex_{stack}$.Push($S_{cur}$)
                end if
            case assume($v == expr$)        ▷ Assume Instruction
                $I_{cur} \leftarrow$ getNextInstruction($I_{cur}$)
                if isSat($v == expr$) then
                    $S_{cur} \leftarrow [I_{cur}, \phi_{path} \wedge (v == expr), \Delta]$
                else
                    $S_{cur} \leftarrow [I_{cur}, \phi_{path} \wedge (\text{false}), \Delta]$
                end if
                $Ex_{stack}$.Push($S_{cur}$)
            case HALT        ▷ Terminate Instruction
                $\varphi_{paths} \leftarrow \varphi_{paths}$.Append($\phi_{path}$)
    end while
return $\varphi_{paths}$
end function
```

- Let $d : \mathit{ForallAssign} \times \mathit{Mems} \to [0,1]$ be a distribution of program memories parameterized by assignments to universally quantified symbolic variables and let $\mathit{MemDists}$ be the set of all parameterized distributions of program memories.

Additionally, we will use emphatic brackets for two purposes:

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

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

- If $e \in \mathit{SymExprs}$ is a *symbolic* expression containing the symbolic variables $\alpha_1, \ldots, \alpha_n \in \mathit{ForallSymVars}$ and $\delta_1, \ldots, \delta_m \in \mathit{ProbSymVars}$, and $a_f \in \mathit{ForallAssign}$ and $a_p \in \mathit{ProbAssign}$, 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 $\varphi : \mathit{ForallAssign} \times \mathit{ProbAssign} \to \{0,1\}$ denotes whether the path condition is true or false under the given assignments, $\sigma : \mathit{Vars} \to \mathit{SymExprs}$ is the mapping from program variables to symbolic expressions generated through symbolic execution, and $P : \mathit{ForallAssign} \to \mathit{ProbSymVars} \rightharpoonup (\mathit{Vals} \to [0,1])$ 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, $a_f \in \mathit{ForallAssign}$, $\text{domain}(P(a_f)) = \{\delta_1, \ldots, \delta_k\}$. Let $\alpha_1, \ldots, \alpha_l \in \mathit{ForallSymVars}$ 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 convertToMem : $(\mathit{Vars} \to \mathit{SymExprs}) \to \mathit{ForallAssign} \to \mathit{ProbAssign} \to \mathit{Mem}$ as

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

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 $d \in \mathit{MemDists}$ 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 $\mathrm{assign}_{\mathsf{x}=e} : \mathit{Mems} \longrightarrow \mathit{Mems}$ 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 $\mathrm{desample} : \mathit{Vars} \times \mathit{Mems} \longrightarrow \mathcal{P}(\mathit{Mems})$ 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 $a_f \in \mathit{ForallAssign}$ be an arbitrary assignment of forall symbolic variables, c be a guard of an `if` condition, $d \in \mathit{MemDists}$ be a distribution of program memories parameterized by assignments to forall symbolic variables, and $\mathsf{x}_1, \ldots, \mathsf{x}_n \in \mathit{Vars}$ be all of the program variables in c. Then for all program memories $m \in \mathit{Mems}$, d conditioned on a guard c being true, represented as d_c is defined as

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

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

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

3.3.2 *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.

KLEE assigns values to these *symbolic* variables by employing an SMT Solver for solving the current set of *constraints* seen so far in the program path as described in Algorithm 4 and also by concretizing some of the values that get evaluated as a result of *concrete* execution (Eg. return value from external function calls). Upon reaching the *branch* instruction at Line 13 in Algorithm 4, 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* and *false* 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. 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*.

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*, (2) *branch*, & (3) *assume* statements. These dumps are later refined and used by the tool for further processing.

5 CASE STUDIES

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!

- Compare and Contrast with Mayhap, PSE, AxProf, P4WN & PSI.
- Explain how we differ by not using Model Counting but a Constraint Solving approach.
- Support for ForAll variables along with *probabilistic* variables.

8 CONCLUSION & FUTURE WORK

ACKNOWLEDGMENTS

A APPENDIX

Text of appendix …