Symbolic Execution for Randomized Programs

ZACHARY SUSAG, Cornell University, USA
SUMIT LAHIRI, IIT Kanpur, India
JUSTIN HSU, Cornell University, USA
SUBHAJIT ROY, IIT Kanpur, India

We propose a symbolic execution method for programs that can draw random samples. In contrast to existing work, our method can verify randomized programs with unknown inputs and can prove probabilistic properties that universally quantify over all possible inputs. Our technique augments standard symbolic execution with a new class of *probabilistic symbolic variables*, which represent the results of random draws, and computes symbolic expressions representing the probability of taking individual paths. We implement our method on top of the KLEE symbolic execution engine alongside multiple optimizations and use it to prove properties about probabilities and expected values for a range of challenging case studies written in C++, including Freivalds' algorithm, randomized quicksort, and a randomized property-testing algorithm for monotonicity. We evaluate our method against Psi, an exact probabilistic symbolic inference engine, and Storm, a probabilistic model checker, and show that our method significantly outperforms both tools.

CCS Concepts: • **Mathematics of computing** → *Probabilistic inference problems*; • **Software and its engineering** → **Software verification**; *Automated static analysis*; • **Theory of computation** → **Automated reasoning**; *Program verification.*

Additional Key Words and Phrases: Probabilistic programming languages, symbolic execution, automated software verification

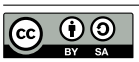
ACM Reference Format:
Zachary Susag, Sumit Lahiri, Justin Hsu, and Subhajit Roy. 2022. Symbolic Execution for Randomized Programs. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 181 (October 2022), 30 pages. https://doi.org/10.1145/3563344

1 INTRODUCTION

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

Researchers have adapted SE to many domains (e.g., [Borges et al. 2014; Farina et al. 2019; Filieri et al. 2013; Geldenhuys et al. 2012; Kang et al. 2021; Kiezun et al. 2012; Sasnauskas et al. 2011, 2010]). In this work, we consider how to perform SE for *randomized* programs, which can draw random samples from built-in distributions. These programs play a critical role in many important applications today, from machine learning to security and privacy. However, randomized programs, like all programs, are susceptible to bugs [Joshi et al. 2019]. Correctness properties are

Authors' addresses: Zachary Susag, Cornell University, USA, zjs@cs.cornell.edu; Sumit Lahiri, IIT Kanpur, India, sumitl@cse.iitk.ac.in; Justin Hsu, Cornell University, USA, justin@cs.cornell.edu; Subhajit Roy, IIT Kanpur, India, subhajit@cse.iitk.ac.in.



2475-1421/2022/10-ART181
https://doi.org/10.1145/3563344

difficult to formally verify; existing methods typically require substantial manual effort and target narrow properties. Moreover, randomized programs are difficult to test: the desired behavior is not deterministic, and it may not be possible to detect that a program is producing an incorrect distribution of outputs running the program.

Challenges and prior work. Accordingly, randomized programs are an attractive target for formal verification methods. However, there are several challenges in developing a SE procedure for randomized programs:

- **Quantitative properties.** Standard SE aims to check whether a bad program state is reachable. In randomized programs, we are often more interested in whether a bad state is reached *too often*, or whether a good state is reached *often enough*. Accordingly, SE for randomized programs must be able to analyze the *quantitative* probability of reaching program states.
- **Computing branch probabilities.** A concrete execution of a non-probabilistic program follows one branch at each branching instruction, since the branch condition is either true or false in any program state. In probabilistic programs, the program can be thought of as transforming a *distribution* of program states. Thus, a branch condition has some probability of being true and some probability of being false in every probabilistic program state. A SE procedure for probabilistic programs should be able to compute probabilities of branches and combine these probabilities to reason about probabilities of whole paths.
- **Handling unknown inputs.** Like standard programs, randomized programs usually accept input parameters. These unknown values are qualitatively different from the unknown results of random sampling commands: there is no probability assigned to different settings of the inputs, and the target correctness property universally quantifies over all possible inputs.
  Prior work has considered SE for probabilistic programs [Geldenhuys et al. 2012; Sampson et al. 2014], but unlike traditional SE, most existing methods do not treat the program inputs as *unknown* values. Instead, existing methods typically consider probabilistic programs without inputs [Sampson et al. 2014] or assume that the inputs are drawn from a known probability distribution [Geldenhuys et al. 2012]. This simplification allows existing SE methods to compute probabilities exactly using model counting or approximately using statistical sampling. However, it significantly limits the kinds of programs and properties that can be analyzed.

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

- **Probability bounds.** For all inputs, the *probability of reaching a program state* is at most/at least/exactly equal to some quantity, which may depend on the inputs. These properties are the probabilistic analog of the reachability properties considered by traditional SE.
- **Expectation bounds.** For all inputs, the *expected value of a program expression* in the output is at most/at least/exactly equal to some quantity, which may depend on the inputs. These properties are useful for bounding expected resource usage or running time.

There are two key technical ingredients in our approach:

- **Distinguishing between regular and probabilistic symbolic variables.** Our method models sampling statements by introducing a new kind of *probabilistic* symbolic variable. While regular symbolic variables represent unknown inputs, probabilistic symbolic variables represent the result from sampling a known distribution.
- **Computing symbolic branch probabilities.** Branch probabilities may depend on unknown program inputs. Accordingly, we cannot use approaches like model counting to compute the

branch probabilities since the probabilities are not constants. Instead, our method computes the branch probability expressions symbolically.

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

- A symbolic execution method for probabilistic programs with unknown input parameters and a formal proof of soundness for our method (§3).
- A collection of case studies from the randomized algorithms literature (§4), including bounding the soundness probability of Freivalds' algorithm [Freivalds 1977], the expected number of comparisons for randomized quicksort, and the correctness probability of a randomized algorithm for checking monotonicity [Goldreich 2017]. All examples have unknown input parameters.
- An optimized implementation of our approach called Plinko (§5). Plinko is built on top of the KLEE symbolic execution engine [Cadar et al. 2008], allowing Plinko to perform symbolic execution on real implementations of randomized programs written in any language that can generate LLVM bytecode, while also faithfully modeling program states on real hardware (e.g., with overflowing arithmetic, arrays, etc.).
- A thorough experimental evaluation of our tool (§6). We show that our method significantly outperforms two well-developed tools for analyzing probabilistic programs: Psi [Gehr et al. 2016, 2020], a leading tool for exact probabilistic symbolic inference, and Storm, a probabilistic model checker [Hensel et al. 2022]. We also evaluate the factors affecting the performance of Plinko and the effectiveness of our optimizations.

We conclude by discussing other potential optimizations (§7), related work (§8), and future directions (§9). Source code for Plinko and our case studies is available on Zenodo [Susag et al. 2022]. The full version of this paper is available on `arXiv`, which contains full proofs and additional details.

2 OVERVIEW

The Monty Hall problem [Selvin 1975] is a classic probability puzzle based on the American television show *Let's Make a Deal*. The problem itself is simple:

> You are a contestant on a game show. 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, behind which is a goat. The host then offers you the choice to switch to the remaining closed door. Should you?

While it may seem counterintuitive, you should always switch doors: regardless of your original door choice, you will win the car two-thirds of the time if you do switch and only a third of the time if you do not. We model the Monty Hall problem as the probabilistic program in Figure 2. The variable `choice` $\in [1, 3]$ represents the door that is originally chosen by the contestant, while the boolean variable `door_switch` represents whether the contestant chooses to switch doors. Regardless of the value of `choice`, if `door_switch == true` then `monty_hall` should return `true` (i.e., the contestant wins) two-thirds of the time.

Our goal is to automate this reasoning. More generally, the problem we aim to solve is: given a probabilistic program with discrete sampling statements and a target probability bound, how do we verify that the program satisfies the bound? While the property for the Monty Hall problem is possible to verify with existing methods due to its small input space (i.e., it is feasible to try all possible inputs and verify the probability bound on each output distribution), more realistic probabilistic programs often have an enormous space of inputs (e.g., randomized quicksort takes an array of integers as input). We solve this problem by developing a symbolic execution method.

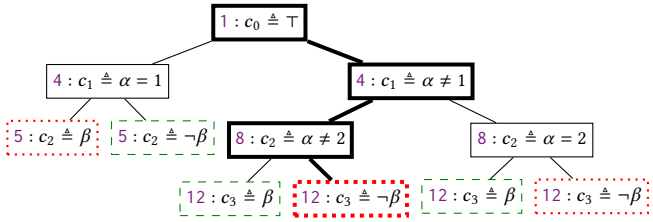


Fig. 1. Symbolic execution tree for the Monty Hall Problem if the car is behind door 1.

In symbolic execution, program inputs are replaced by *symbolic* variables that can take on any value. The program is then "run" on these symbolic variables. When a branch is encountered (e.g., an `if` condition), execution proceeds along both branches, and the branch's guard is recorded in that path's *path condition*, denoted by φ. This yields a *symbolic execution tree* that describes all possible paths through the program.

To illustrate this idea, suppose we were to use traditional symbolic execution to analyze the program in Figure 2 but fixed the car behind door 1. The two inputs, `choice` and `door_switch`, would become the symbolic variables α and β respectively. The execution tree is shown in Figure 1. (Symmetric trees can be made for the cases when the car is behind doors 2 and 3.). Each node contains the corresponding line number and branch guard. Leaves surrounded by a green dashed line (- -) denote paths where the program returns `true` (i.e., the contestant wins the car), and leaves surrounded by a red dotted line (....) denote paths where the program returns `false` (i.e., the contestant wins a goat). A path condition φ is a conjunction of all the c_i between the root of the tree and a leaf.

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

Fig. 2. C code for the Monty Hall problem.

For example, consider the bolded path in Figure 1. Execution begins on line 1 with an empty path condition: $\varphi = \top$. When the first `if` condition is reached on line 4, `if (choice == car_door)`, the path follows the false branch. We then conjoin the corresponding symbolic expression to φ, i.e., $\varphi = \alpha \neq 1$. Execution along this path then jumps to line 6. However, the guard `choice != 1 && car_door != 1` is always false as `car_door == 1`. Since there is no ambiguity about which branch to follow, φ remains unchanged. The next branch is on line 8, which is equivalent to checking whether `car_door != 2`. The path takes the true branch, so we update φ with the new guard: $\varphi = (\alpha \neq 1) \wedge (\alpha \neq 2)$. The last branch is on line 12, where the program checks whether the contestant chose to switch doors. This guard ultimately determines whether the contestant wins the car along this path as we previously determined that the contestant did not originally pick the winning door. Here, switching wins the car while not switching only wins a goat. The entire path condition φ can then be written as $\varphi = (\alpha \neq 1) \wedge (\alpha \neq 2) \wedge \neg\beta$.

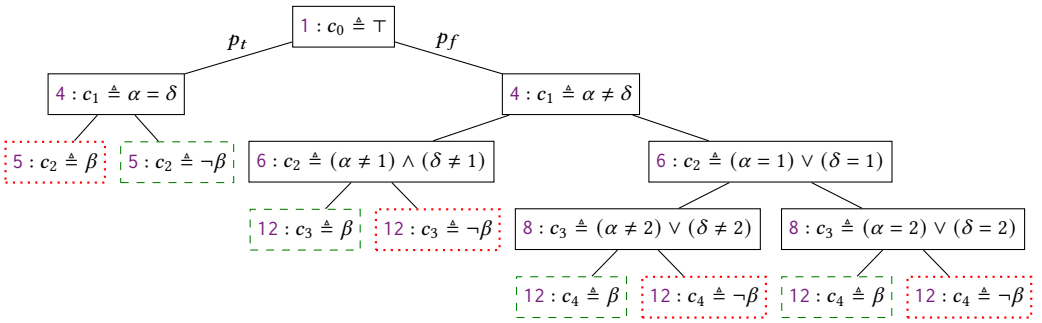


Fig. 3. Symbolic execution tree for the Monty Hall Problem with probabilistic symbolic variables. p_t and p_f denote the probability expressions for taking the true and false branches of the `if` condition found on line 4 of Figure 2. While every branch has an associated probability expression, we omit displaying them for presentation purposes.

Given a setting of the input parameters, this analysis can tell us whether the contestant will win the car when the car is behind door 1; however, it cannot tell us how likely winning the car is. In principle, if we combined the tree shown in Figure 1 with the symmetric trees for the cases where the car is behind doors 2 and 3, and if we knew how likely it was to take one branch over another, we could extend this reasoning to determine how often we would reach a winning state.

To carry out this kind of quantitative reasoning within symbolic execution, **our core idea is to represent probabilistic samples as a new class of symbolic variables.** We refer to these variables as *probabilistic* symbolic variables in contrast to regular symbolic variables, which we instead call *universal* symbolic variables. Just as in standard symbolic execution, universal symbolic variables represent unknown program inputs. Since these program inputs are universally quantified instead of being drawn from some assumed distribution, the name "universal" symbolic variable is appropriate. In contrast, probabilistic symbolic variables represent the result of random draws. During symbolic execution, instead of actually drawing a random sample, we represent the would-be sample as a probabilistic symbolic variable and record the source distribution. Our method uses this information to compute the (symbolic) probabilities of taking branches.

To demonstrate, let `car_door` be represented by the probabilistic symbolic variable, δ. Instead of branching solely on the universal symbolic variables α and β, we additionally branch on the probabilistic symbolic variable δ. This execution tree is presented in Figure 3. We elide branches where the current path condition shows that only one choice is feasible.

Since probabilistic symbolic variables are drawn from a distribution, we can compute the *probability* of taking a certain branch by simply counting how many values from the domain of the distribution satisfy the guard condition. For example, to compute the probability of taking the true branch in line 4 (denoted by p_t in Figure 3), it suffices to count how many settings of δ satisfy $\alpha = \delta$ and divide by the total number of possible assignments to δ, or 3, as `car_door` is *uniformly* randomly chosen to be 1, 2, or 3. However, since α is a symbolic variable, we can only obtain a probability *expression* over universal symbolic variables. Let $[\cdot]$ denote Iverson brackets, where $[Q] = 1$ if formula Q is true and 0 otherwise. Then, $p_t \triangleq ([\alpha = 1] + [\alpha = 2] + [\alpha = 3])/3 = 1/3$ as $\delta \in \{1, 2, 3\}$ and each setting is equally likely. Similarly, $p_f \triangleq ([\alpha \neq 1] + [\alpha \neq 2] + [\alpha \neq 3])/3 = 2/3$. We stress that in general these probabilities can be *symbolic expressions* as they may depend on the universal symbolic variables. This feature is one of the key advantages of our technique over existing methods for probabilistic symbolic execution: it makes reasoning about the probabilities much more complex, but it also allows us to prove properties for programs with truly unknown inputs. In §3, we explain how our technique can automatically generate these probability expressions and,

by using standard probability rules, combine them to construct probability expressions for entire paths.

Returning now to our motivating question: if the contestant switches doors, do their chances of winning the car exceed 1/3? Using the tree in Figure 3, we can limit our focus to just those paths that lead to a win. Then, to find the probability of winning the car given the contestant switches doors, we filter out those paths where $\neg\beta$ is true and sum together the probabilities of the remaining paths. Regardless of what α is, the resulting expression produces a probability of 2/3.

3 PROBABILISTIC SYMBOLIC EXECUTION

Now that we have described how our approach works at a high level, we can flesh out the details. We begin with preliminary details (§3.1) and an overview of standard symbolic execution for non-probabilistic programs (§3.2). Then, we present our symbolic execution approach for probabilistic programs. Our approach has two steps: first, we augment standard symbolic execution to track probabilistic information as paths are explored (§3.3). Then, we convert the program paths into a logical query that encodes a probabilistic property (§3.4). Finally, we formalize our approach mathematically and prove its soundness (§3.5).

3.1 Preliminaries

Our algorithm works on programs written in **pWhile**, a core imperative probabilistic programming language. The statements of **pWhile** are described by the following grammar:

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

Intuitively, the assignment statement $\mathsf{x} \leftarrow e$ assigns the result of evaluating the expression e to the program variable x, while the sampling statement $\mathsf{x} \sim d$ draws a random sample from a distribution d and assigns the result to the program variable x. Here, d is a *discrete* distribution expression defining which distribution the sample should be drawn from; these distributions are primitives, corresponding to mathematically standard distributions (e.g., uniform or coin flip distributions). We interpret distributions as functions from values to the range $[0, 1]$ (i.e., the probability of the value occurring in the distribution). For example, $\mathsf{UniformInt}(1, 6)$ is a uniform distribution that selects a random integer between 1 and 6 (inclusive). For presentation purposes, we limit our focus to uniform integer distributions, but our method can support other finite, discrete distributions, such as the Bernoulli or uniform joint distributions.

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

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

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

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

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

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

3.2 Symbolic Execution

Algorithm 1 Probabilistic Symbolic Execution Algorithm

```
1: procedure SymbEx(Prog : Program, x⃗ : Program Inputs, ψ : Predicate)
2:   E_s ← [], σ_init ← BindToSymbolic(x⃗), Enc_ψ ← 0.0, Enc_v ← 0.0
3:   I_0 ← getStartInstruction(Prog)
4:   S_0 ← (I_0, ⊤, σ_init, ∅, 1.0)
5:   E_s.Push(S_0)
6:   while E_s ≠ ∅ do
7:     (I_c, φ, σ, P, p) ← E_s.Pop( )
8:     if UNSAT(φ) then
9:       continue
10:    switch instType(I_c) do
11:      case [x ← e]
12:        I_1 ← getNextInstruction(I_c)
13:        σ[x] ← σ[e]
14:        S_1 ← (I_1, φ, σ, P, p)
15:        E_s.Push(S_1)
16:      case [x ~ d]
17:        I_1 ← getNextInstruction(I_c)
18:        δ ← Generate a fresh probabilistic symbolic variable
19:        σ_1, P_1 ← (σ[x ↦ δ], P[δ ↦ d])
20:        S_1 ← (I_c, φ, σ_1, P_1, p)
21:        E_s.Push(S_1)
22:      end case
23:      case [if c then goto T]
24:        c_sym ← σ[c]
25:        p_t, p_f ← PBranch (c_sym, φ, P)
26:        I_1 ← getInstruction(T)
27:        I_2 ← getNextInstruction(I_c)
28:        S_t ← (I_1, φ ∧ c_sym, σ, P, p · p_t)
29:        S_f ← (I_2, φ ∧ ¬c_sym, σ, P, p · p_f)
30:        E_s.Push(S_f)
31:        E_s.Push(S_t)
32:      case [halt]
33:        ψ_sym ← σ[ψ]
34:        if SAT(ψ_sym) then
35:          p_t, p_f ← PBranch(ψ_sym, φ, P)
36:          Enc_ψ ← Enc_ψ + p
37:        end if
38:        Enc_v ← Enc_v + p · σ[v]
39:  r ← Solve (Enc_ψ, Enc_v)
40:  return r
```

Symbolic execution [King 1976] is a program analysis technique that iteratively explores the set of paths through a given program. Since program paths may depend on potentially unknown input variables, symbolic execution treats each input as a *symbolic* variable. All program operations are redefined to manipulate these symbolic variables. We present a high-level description of symbolic execution in Algorithm 1. The reader should ignore the portions in shaded boxes for now; these are our extensions to handle probabilistic programs.

Symbolic execution tracks program state using the following data structures: the next instruction I_c; a *path condition* φ, which is a conjunctive formula consisting of the symbolic branch constraints that are true for a particular path; and an *expression map* σ, which maps program variables to

symbolic expressions over constants and symbolic variables. To begin, Algorithm 1 initializes the execution stack E_s with an empty list and the expression map σ with fresh symbolic variables for each *program input* in the input vector $\vec{x}$ by calling BindToSymbolic, as shown on line 2. I_0 is then initialized with the first instruction of the program on line 3, and the initial state tuple S_0 is created on line 4 and appended to E_s.

The core of Algorithm 1 iterates through all reachable states until E_s is exhausted (lines 6 to 38). For each iteration, we pop a state from E_s (line 7), and, if the selected path condition φ is feasible, the target instruction is analyzed based on the instruction's form: line 11 for assignment statements, and line 23 for branch statements.

For an assignment statement $\mathsf{x} \leftarrow e$ where e is a program expression, we first convert e into a *symbolic* expression using the expression map σ. We use the notation $\sigma[e]$ to denote the symbolic expression e_{sym} constructed by replacing each program variable in e with its corresponding symbolic expression in σ. For a branch statement **if** c **then goto** T, Algorithm 1 *forks* by constructing the corresponding path conditions for both branches, pushing the symbolic branch guard c_{sym} in the positive and negative form (for the true and false branches, respectively) to E_s. Finally, symbolic execution along a path terminates at a **halt** instruction, and information about the final symbolic state is collected for subsequent analysis.

3.3 Probabilistic Symbolic Execution

To support random sampling instructions, our algorithm distinguishes between two categories of symbolic variables:

- *Universal symbolic variables*: These are identical to those in traditional symbolic execution and are used to model unknown program inputs.
- *Probabilistic symbolic variables*: These model a random sample from a known distribution. Fresh probabilistic symbolic variables are created for each sampling statement to represent the result of a random draw.

Symbolic execution becomes more challenging when both universal and probabilistic variables are present as branches are taken with some quantitative probability dependent on the results of the random samples and the universal symbolic variables. To track information about probabilistic states, we use the following data structures:

- *Distribution map* (P): Analogous to σ, this map from probabilistic symbolic variables to distribution expressions tracks the distribution from which a probabilistic symbolic variable was originally sampled from.
- *Path probability* (p): For each path, we maintain a path probability expression p over the universal symbolic variables.

We can now define how symbolic execution handles sampling statements and how to compute the probability of taking either side of a branch statement.

Sampling (Lines 16 to 22). On encountering a sampling statement $\mathsf{x} \sim d$ (line 16 of Algorithm 1), we generate a fresh probabilistic symbolic variable δ to model the random value sampled from the distribution described by the distribution expression d. We update the expression map σ to map the program variable x to δ and record δ's distribution in the distribution map P.

Branches (Lines 23 to 31). Similar to traditional symbolic execution, upon reaching a branch statement our algorithm pushes the symbolic states corresponding to the true and false branches to E_s. Then, PBranch (Algorithm 2) computes probability *expressions* for the true and false branches. PBranch takes the following as inputs: an expression c_{sym}, which is the symbolic equivalent of the program guard expression c; the current path condition φ; and the current distribution map

Algorithm 2 Probabilistic Branch Algorithm

1: **function** PBranch(c_{sym}, φ, P)
2: $(\delta_1, \ldots, \delta_n) \leftarrow \text{dom}\,(P)$
3: $(d_1, \ldots, d_n) \leftarrow (P[\delta_1], \ldots, P[\delta_n])$
4: $\mathcal{D} \leftarrow \text{dom}\,(d_1) \times \cdots \times \text{dom}\,(d_n)$
5: $p_c \leftarrow \dfrac{\sum_{(v_1,\ldots,v_n)\in\mathcal{D}} [(c_{sym} \wedge \varphi)\{\delta_1 \mapsto v_1, \ldots, \delta_n \mapsto v_n\}]}{\sum_{(v_1,\ldots,v_n)\in\mathcal{D}} [\varphi\{\delta_1 \mapsto v_1, \ldots, \delta_n \mapsto v_n\}]}$
6: $p'_c \leftarrow 1 - p_c$
7: **return** (p_c, p'_c)

P. It returns two probability expressions (p_c, p'_c) representing the probabilities of taking the true and false branches respectively. For simplicity, we present this subroutine in the simplified setting where all sampling instructions are from discrete uniform distributions; handling general weighted distributions is not much more complicated and is supported in our implementation.

To gain intuition for how we compute path probabilities, consider the case where the guard condition contains *only* universal symbolic variables. Let $c_{sym} = \sigma[c]$ be the equivalent symbolic expression for the guard c and assume that c_{sym} does not mention any probabilistic symbolic variables. In this case, the branch that is taken is entirely determined by the universal symbolic variables. For a fixed setting of the universal symbolic variables, one branch must have a probability of 1 and the other 0. Put another way, either c_{sym} or $\neg c_{sym}$ is satisfiable but never both. We use Iverson brackets to represent the symbolic probability of taking the "true" and "false" branches, namely $[c_{sym}]$ and $[\neg c_{sym}]$ respectively.

For probabilistic branches—guards which contain probabilistic symbolic variables—computing the branch probability is more involved. We give an intuitive justification here and defer the proof of correctness to §3.5. When Algorithm 1 encounters a branch statement, recall that φ contains the necessary constraints on the symbolic variables that must hold in order to reach the branch statement in question. Therefore, we can view the probability of taking either branch as a *conditional probability*: the probability that c_{sym} holds assuming that φ is satisfied. Formally, we want to compute $\Pr[c_{sym} \mid \varphi] = \Pr[c_{sym} \wedge \varphi]/\Pr[\varphi]$.

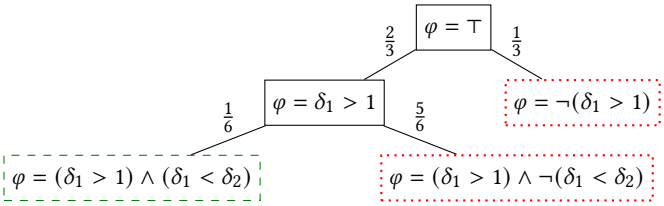
Algorithm 2 builds p_c, a formula for $\Pr[c_{sym} \mid \varphi]$. Note that each probabilistic symbolic variable δ is mapped to precisely one distribution, d. Assuming there are n probabilistic symbolic variables $\delta_1, \ldots, \delta_n$ and n distributions $d_1, \ldots, d_n$ the set of all possible values for $\delta_1, \ldots, \delta_n$ is $\mathcal{D}$ (line 4). Using our simplifying assumption that all probabilistic symbolic variables are drawn from uniform distributions, each assignment $(v_1, \ldots, v_n)$ for $(\delta_1, \ldots, \delta_n)$ has equal probability. Thus, instead of computing the conditional probability as a ratio of probabilities, we calculate the ratio of the number of assignments from $\mathcal{D}$ which satisfy $c_{sym} \wedge \varphi$ and φ as shown on line 5 of Algorithm 2. We stress that p_c is a *symbolic expression* possibly containing universal symbolic variables. Finally, we use the fact that the sum of the conditional probabilities of the branch outcomes is 1 to get the probability of the false branch.

We can now compute the probability of an entire path $\varphi \wedge c_{sym}$ (or $\varphi \wedge \neg c_{sym}$). Since p_t is the *conditional* probability of c_{sym} being true (line 25) and p is the probability of φ, we can use the standard rule $\Pr[A \wedge B] = \Pr[A \mid B] \Pr[B]$, where A and B are any two events, to conclude that $\Pr[\varphi \wedge c_{sym}] = p \cdot p_t$. The same can be done for $\varphi \wedge \neg c_{sym}$ as shown on lines 28 and 29.

Example. Consider the program in Figure 4a, and suppose we wish to calculate the probability of the program returning `True`. Following Algorithm 1, we generate fresh probabilistic symbolic

```
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) Example program.



(b) Symbolic execution tree.

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

variables δ_1 and δ_2 for x and y respectively, and record their originating distribution $\mathcal{U}\{1,3\}$. Using Algorithm 2 to process line 4 of Figure 4a, note that $\mathcal{D} = \{1,2,3\} \times \{1,2,3\}$ and

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

Therefore, the probability of taking the true branch of the inner **if** condition is only 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 setting of x and y will satisfy (x > 1) ∧ (x < y), namely x = 2 and y = 3. We can use Algorithm 2 on the remaining branches in Figure 4a to yield the fully annotated symbolic execution tree in Figure 4b.

3.4 Query Generation

Recall that our original goal was to prove bounds on the probability that a program Prog terminates in a state where an arbitrary predicate ψ holds. We frame this question as a logical query using the symbolic state found in the execution tree. In general, our queries are of the following form:

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

where $\alpha_1, \ldots, \alpha_n$ are *all* the universal symbolic variables found in Prog, Enc_ψ is an expression representing the probability that ψ holds in Prog, $\bowtie$ is a binary relation (e.g., $>, <, \geq, \leq$), and f is some lower/upper bound function of the universal symbolic variables that we want to prove Prog does not violate. Another way of interpreting f is a function which computes the *desired* probability of ψ occurring in Prog (e.g., a maximum acceptable error rate) given a setting of the universal symbolic variables.

We construct the expression Enc_ψ on lines 33 to 38 of Algorithm 1. Once a path reaches a **halt** statement, ψ is converted into its corresponding symbolic expression ψ_{sym} using the expression map σ. If ψ_{sym} is satisfiable, we use Algorithm 2 to calculate the probability of ψ_{sym} being true given φ, denoted by p_ψ. We then add $p \cdot p_\psi$, the probability that we traversed φ and ψ_{sym} is true along path φ, to the current expression Enc_ψ. Note that some settings of the universal symbolic variables, $\alpha_1, \ldots, \alpha_n$, may not be reachable along path φ. Since Algorithm 2 embeds the path condition φ into the probability expressions (p_t, p_f), if φ is falsified, then the probability of φ is 0. Once all reachable states in E_s are exhausted, the total probability that ψ holds for all reachable paths in

Prog is Enc_ψ. Solve is then called on line 39, which takes in Enc_ψ, constructs (1), and calls and automated decision procedure, such as an SMT solver, to answer the query.

The meaning of (1) depends on the termination condition of Algorithm 1. If all reachable paths are explored, then Enc_ψ represents the exact probability that ψ holds in the output distribution, parameterized by the unknown input variables. However, we will not always be able to explore all paths. In this case, Enc_ψ may not be equal to the true probability of ψ, but it is always a sound *lower bound* of the true probability. While this may not be enough to verify typical correctness properties, it can be useful for *refuting* certain kinds of correctness properties. For instance, if we want to check that the probability of ψ is at most $f(\alpha_1, \ldots, \alpha_n)$ for all settings of the input variables, and Algorithm 1 produces a probability mass Enc_ψ that exceeds $f(\alpha_1, \ldots, \alpha_n)$ for some setting of the input variables, then the original upper bound cannot hold.

Example. In §2 we introduced the Monty Hall problem. We wanted to prove that, if a contestant always switched doors, their probability of winning the car is $2/3$. We can frame this property as a query of the form defined in (1). If α is the universal symbolic variable corresponding to `choice`, and β is the universal symbolic variable corresponding to `door_switch`, let $\psi \triangleq \beta \wedge \mathsf{monty\_hall}(\alpha, \beta)$, $\bowtie \triangleq =$, and $f \triangleq 2/3$. Then, our full query is $\forall \alpha, \beta \,.\, Enc_\psi = 2/3$. Algorithm 1 then constructs the tree found in Figure 3 and the probability expression for each path. If φ_i is the path condition for the path ending in the ith-left-most leaf of Figure 3 and p_i is the probability expression for φ_i as created by Algorithm 1, then $Enc_\psi = p_3 + p_5 + p_7$. Lastly, Solve takes the query $\forall \alpha, \beta \,.\, p_3 + p_5 + p_7 = 2/3$ and uses an external automated decision procedure to prove for each possible setting of α and β that $p_3 + p_5 + p_7 = 2/3$.

3.4.1 Expected Value Queries. In addition to probability-bound queries, our technique can also prove bounds on the expected value of any variable with only minor modifications. Suppose that we want to find the expected value of an arbitrary program variable named v, denoted $\mathbb{E}[\mathsf{v}]$. Since each path has a probability p_i and an expression map σ_i, we can construct a symbolic expression that computes $\mathbb{E}[\mathsf{v}]$, namely $\sum_{i=1}^{n} p_i \cdot \sigma_i[\mathsf{v}]$, as $\sigma_i[\mathsf{v}]$ is the symbolic expression which is the value of v along the i^{th} path (line 38 of Algorithm 1). Then, for Solve on line 39, we use an automated tool to solve the query

$$\forall \alpha_1, \ldots, \alpha_n \,.\, Enc_{\mathsf{v}} \bowtie f(\alpha_1, \ldots, \alpha_n)$$

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

3.5 Formalization

In this section, we state a series of soundness theorems for Algorithm 1.

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

Lemma 3.1 (Equivalency between p and (φ, P)). *For all program statements, S in **pWhile**, Algorithm 1 maintains*

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

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

We prove soundness for Algorithm 1 by showing that our abstraction R respects the semantics for each of the three main types of statements in **pWhile**, namely *assignment*, *sampling*, and *branching*.

Theorem 3.2 (Correctness of Assignment (Lines 13 to 14)). *If a distribution* μ *satisfies* $R = (\varphi, \sigma, P)$ *and Algorithm 1 reaches an assignment statement of the form* $\mathsf{x} \leftarrow e$, *then* $\mu_{\mathsf{x} \leftarrow e}$, *the distribution of program memories after executing* $\mathsf{x} \leftarrow e$, *satisfies* $R' = (\varphi, \sigma', P)$.

Theorem 3.3 (Correctness of Sampling (Lines 19 to 20)). *If a distribution* μ *satisfies* $R = (\varphi, \sigma, P)$ *and Algorithm 1 reaches a sampling statement of the form* $\mathsf{x} \sim d$, *then* $\mu_{\mathsf{x} \sim d}$, *the distribution of program memories after executing* $\mathsf{x} \sim d$, *satisfies* $R' = (\varphi, \sigma', P')$.

Theorem 3.4 (Correctness of Branching (Lines 25, 28 and 29)). *If a distribution* μ *satisfies* $R = (\varphi, \sigma, P)$, *and Algorithm 1 encounters a branching statement of the form* **if** c **then goto** T, *then* μ_c *satisfies* $R_{true} = (\varphi \wedge c_{sym}, \sigma, P)$ *and* $\mu_{\neg c}$ *satisfies* $R_{false} = (\varphi \wedge \neg c_{sym}, \sigma, P)$ *if all the distributions in* P *are uniform. Additionally, for all* $a_\forall \in A_\forall$,

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

where $p'_c = 1 - p_c$.

4 CASE STUDIES

In this section, we introduce the case studies that we will use in our evaluation. For each case study, we give a brief description of the algorithm and the target property we wish to verify. Additionally, we specify which variables are *concretized* for our evaluation. Like other symbolic execution methods, our tool requires fixing, or concretizing, some parameters, such as the lengths of input arrays and loop bounds. We discuss in greater depth why concretization is necessary and discuss the implications it has on performance in §6.

Freivalds' Algorithm. Freivalds' algorithm [Freivalds 1977] is a randomized algorithm used to verify matrix multiplication. Given three $n \times n$ matrices A, B, and C, if $A \times B = C$, then the algorithm always returns `true`. However, if $A \times B \neq C$, the probability that Freivalds' algorithm returns `true` (i.e., a false positive) is at most $1/2$.

We want to verify that the false positive rate is at most $1/2$. Letting

$$\psi \triangleq A \times B \neq C \wedge \mathsf{Freivalds}(A, B, C, n) = \mathtt{true},$$

our query is then:

$$\forall A, B, C \;.\; Enc_\psi \leq 1/2$$

where n is concretized. To reduce the probability of false positives, Freivalds' algorithm can be run k times, returning `true` only if all calls return `true`, and `false` otherwise. Since each trial is independent, the probability of a false positive given that $A \times B \neq C$ is at most $(1/2)^k$. If we let $\psi \triangleq A \times B \neq C \wedge \mathsf{MultFreivalds}(A, B, C, n, k) = \mathtt{true}$, our query then becomes: $\forall A, B, C \;.\; Enc_\psi \leq (1/2)^k$, where n and k are concretized.

The assertion $A \times B \neq C$ can be encoded as the following logical formula, SomeOff:

$$\textsc{SomeOff} \triangleq \bigvee_{i=1}^{n} \bigvee_{j=1}^{n} (A \times B)_{i,j} \neq C_{i,j}$$

which asserts that *at least* one element differ between $A \times B$ and C.

We also consider two other encodings:

$$\textsc{AllOff} \triangleq \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} (A \times B)_{i,j} \neq C_{i,j} \qquad \textsc{FirstOff} \triangleq (A \times B)_{0,0} \neq C_{0,0}$$

AllOff states that *every* element of $A \times B$ must be different from the corresponding element in C, while FirstOff states that the first element of each matrix is different. These conditions imply SomeOff, so they give weaker guarantees: a program that satisfies the specification under AllOff or FirstOff might fail to satisfy the specification under SomeOff. However, if a program *fails* to satisfy the specification under AllOff or FirstOff, then they must also fail to satisfy the specification under SomeOff. These different encodings impact the performance of our tool; we return to these details in §6.

Reservoir Sampling. Reservoir Sampling [Vitter 1985] is a classical, online algorithm which produces a simple random sample S of k elements drawn from a population A of n elements. We want to verify that the random samples returned by reservoir sampling are indeed *simple*, or that each element of A has equal probability of being in S. To make this property easier to state, we consider an equivalent property: if we assume all elements of A are distinct, then the probability that any element of A is in S is exactly k/n. We assume that the elements of A are distinct because if A contained duplicate elements, then it is only true that the probability that each element of A is in S is *at least* k/n, which does not necessarily imply that S is a simple random sample of A.

We frame this property by asking what is the probability that the first element of A appears in S. Let $\psi \triangleq A[1] \neq \cdots \neq A[n] \wedge A[1] \in \mathsf{ReservoirSample}(A, k)$. We then take the query:

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

where n and k are concretized.

Randomized Monotonicity Testing. Randomized monotonicity testing [Goldreich 2017] is a binary-search-like algorithm to determine how far a function $f : [n] \to R$ is from being monotone increasing, where $[n] = \{1, \ldots, n\}$ and R is some ordered set. If f is monotone, then the algorithm will return `true`; if f is δ-*far* from monotone, it will reject with probability greater than δ [Goldreich 2017], where δ-far means that the value of f needs to be changed on at most $\delta \cdot n$ points in order to arrive at a monotone function. Let $\mathsf{DistToMono}(f)$ be the number of elements whose order needs to change to make f monotone, $R = \mathbb{Z}$, and $\psi \triangleq \mathsf{MonotoneTest}(f, n) = \texttt{false}$. Our query is:

$$\forall f : [n] \to \mathbb{Z} \,.\, Enc_{\psi} > \frac{\mathsf{DistToMono}(f)}{n}.$$

where n is concretized, and we universally quantify over all functions with domain $[n]$ and codomain $\mathbb{Z}$.

Randomized Quicksort. Quicksort [Hoare 1961] is a popular sorting algorithm which uses partitioning in order to achieve efficient sorting. While there are many ways of choosing a pivot element, one effective way is randomly. Using this pivot method, the *expected* number of comparisons required is approximately $1.386n \log_2(n)$ where n is the length of the array. We use the method described in §3.4.1 to compute $\mathbb{E}\,[\texttt{num\_comps}]$, where `num_comps` is a program variable which is initially set to 0 and is incremented whenever a comparison occurs for a concretized array length, n.

Bloom Filter. A Bloom filter [Bloom 1970] is a space-efficient, probabilistic data structure used to rapidly determine whether an element is in a set by allowing for false positives. Bloom filters are generally parameterized by the expected number of *unique* elements to be inserted, m, and a *target* maximum false positive error rate after m insertions, ε. We want to verify that after inserting m elements, the actual probability of a false positive is at most ε.

To capture this property, let $x_1, \ldots, x_m$ be m unique elements to be inserted into a Bloom filter B. Note that a false positive occurs when $\mathsf{BloomCheck}(B, y)$ returns `true` when $y \neq x_1 \wedge \cdots \wedge y \neq x_m$ given that and $x_1, \ldots, x_m$ were all previously inserted into B. Therefore, let y be such that $y \neq x_1 \wedge \cdots \wedge y \neq x_m$, and

$$\psi \triangleq B = \mathsf{BloomCreate}(m, \varepsilon) \wedge \left(\bigwedge_{i=1}^{m} \mathsf{BloomInsert}(B, x_i) \right) \wedge \mathsf{BloomCheck}(B, y) = \texttt{true}$$

and take the query to be $\forall x_1, \ldots, x_m, y \,.\, Enc_\psi \leq \varepsilon$, where m and ε are concretized.

Count-min Sketch. A count-min sketch [Cormode and Muthukrishnan 2004] is another space-efficient, probabilistic data structure which encodes a frequency table for a stream of data. To reduce the space usage, the counts can be approximate: the *reported* estimate for the frequency of an element, x, namely $\hat{a}_x$, has the property that $\hat{a}_x \leq \varepsilon \cdot N$ with probability $1 - \gamma$, where $N = \sum_x a_x$, or the total number of elements seen in the sketch, and a_x is the actual count for x. To appropriately initialize the count-min sketch, most implementations are parameterized by the additive error factor ε and the error probability γ.

We want to prove that after inserting n distinct elements, the probability that the error is greater than the additive bound ε is at most $1 - \gamma$. Assuming distinct elements simplifies the query, since the actual counts for each element in the count-min sketch is exactly 1. It is possible to remove this assumption, adjusting the following query to refer to the actual counts for each inserted element.

Similar to how we framed the Bloom filter query, let $x_1, \ldots, x_n$ be n unique elements to be inserted into a count-min sketch C, parameterized by ε and γ. If we let

$$\psi \triangleq C = \mathsf{SketchCreate}(\varepsilon, \gamma) \wedge \left(\bigwedge_{i=1}^{n} \mathsf{SketchUpdate}(C, x_i) \right) \wedge \mathsf{SketchEstimate}(C, x_1) > 1 + \varepsilon \cdot n,$$

then the query becomes

$$\forall x_1, \ldots, x_n \,.\, Enc_\psi \leq \gamma$$

where n, ε, and γ are concretized parameters.

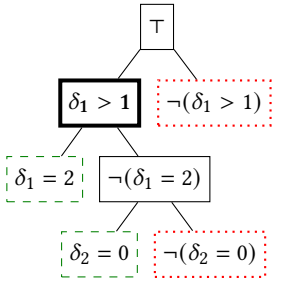
5 IMPLEMENTATION

We implemented our method in a prototype tool called Plinko, building on top of the KLEE symbolic execution engine [Cadar et al. 2008].[1] Plinko constructs the queries described in §3.4, and then dispatches them to the Z3 SMT solver [de Moura and Bjørner 2008]; we use the array, bit-vector, and real number theories from SMT-LIB [Barrett et al. 2016]. If Plinko is unable to prove the query, it returns a counterexample consisting of an input along with the computed probability of the target event. By using KLEE, Plinko is able to directly analyze LLVM bytecode with primitive calls for random sampling. Thus, Plinko can analyze any program written in any programming language that compiles down to LLVM intermediate representation (IR). We require that the end-user annotate which variables should be universally quantified and which are probabilistic, along with the associated distribution, using KLEE intrinsic functions.

[1] Source code for Plinko and all case studies are freely available on Zenodo [Susag et al. 2022]

```
1: x ~ UniformInt(1, 3)
2: if x > 1 then
3:   if x = 2 then
4:     return True
5:   else
6:     y ~ UniformInt(0, 1)
7:     return y = 0
8: else
9:   return False
```

(a) Program.



(b) Symbolic execution tree.

Fig. 5. An example program and symbolic execution tree which exhibits path overlap.

We have also implemented several optimizations for Plinko, which aim to decrease the size and complexity of Enc_ψ, or the expression denoting the probability that the predicate ψ is true. Specifically, we have designed three optimizations: precomputing the probability for simple guards, applying algebraic simplifications, and compressing formulas by factoring out shared terms. We discuss these optimizations here and in the next section (§6) evaluate their effectiveness on our case studies.

Simple Guards. We define a *simple guard* as a predicate of the form $\delta \bowtie k$, where δ is a probabilistic symbolic variable, $P[\delta]$ is a uniform integer distribution, $\bowtie \in \{<, >, \leq, \geq, =, \neq\}$, and k is a constant. For such guards, we can compute the probability of $c = (\delta \bowtie k)$ being true in constant time as there exist closed-form solutions for the number of satisfying assignments to δ which make c true. This effectively allows us to avoid using Algorithm 2 to compute p_c when c_{sym} is a simple guard.

For example, reconsider the outermost conditional statement (line 3) presented in Figure 4a: $\delta_1 > 1$ where $\delta_1 \sim \mathcal{U}\{1, 3\}$. By default, Plinko would use Algorithm 2 to construct $p_{\delta_1 > 1} = [1 > 1] + [2 > 1] + [3 > 1]/3$. This symbolic expression then gets simplified by Z3 to the constant $2/3$. However, for $\delta_1 > 1$, the numerator is just $\max\{\text{dom}\,(\mathcal{U}\{1, 3\})\} - 1 = 2$, and so the probability of $\delta_1 > 1$ being true is simply $2/3$. We have encoded these rewriting rules into Plinko for each primitive binary relation.

Algebraic Simplifications. In order to reduce the work that Z3 must do, we can apply elementary algebraic simplifications on Enc_ψ. For example, we leverage the associative, commutative, distributive properties of addition and multiplication, along with basic boolean algebra laws such as the associativity and commutativity of conjunction and disjunction and De Morgan's laws. We apply a simple fixed-point algorithm to apply each of these laws on Enc_ψ as a preprocessing step before solving the query.

Formula Sharing. The encoded probability Enc_ψ generated by our method is a sum of expressions, one for each explored path. Since paths come from the same symbolic execution tree, terms representing branch probabilities can be repeated multiple times in Enc_ψ across different paths with a shared prefix. These repeated terms can be quite large and complex. Our optimization factors out common terms in Enc_ψ by creating additional logical variables.

To gain intuition, consider the example program in Figure 5. There are two paths where the program in Figure 5a returns `true`, namely $\varphi_1 = (\delta_1 > 1) \wedge (\delta_1 = 2)$ and $\varphi_2 = (\delta_1 > 1) \wedge \neg(\delta_1 = 2) \wedge (\delta_2 = 0)$. Note that φ_1 and φ_2 share a common conjunct, namely $\delta_1 > 1$ which is the bolded node in Figure 5b. By default, Algorithm 1 would construct

$$Enc_\psi = \Pr[\delta_1 > 1] \cdot \Pr[\delta_1 = 2 \mid \delta_1 > 1] + \Pr[\delta_1 > 1] \cdot \Pr[\delta_1 \neq 2 \mid \delta_1 > 1] \cdot \Pr[\delta_2 = 0 \mid \delta_1 \neq 2 \wedge \delta_1 > 1].$$

However, since $\delta_1 > 1$ is a shared conjunct, we are able to factor out $\Pr[\delta_1 > 1]$ from both probability expressions by creating a fresh variable, $p_1 = \Pr[\delta_1 > 1]$ and replacing every occurrence of $\Pr[\delta_1 > 1]$ in Enc_ψ with p_1. We take this idea one step further by observing that $\delta_1 = 2$ appears in φ_1, while its negation appears in φ_2. Since for any event A, $\Pr[A] = 1 - \Pr[\neg A]$, we can set $p_2 = \Pr[\delta_1 = 2 \mid \delta_1 > 1]$ and again apply a substitution on Enc_ψ, resulting in

$$Enc_\psi = p_1 \cdot p_2 + p_1 \cdot (1 - p_2) \cdot \Pr[\delta_2 = 0 \mid \delta_1 \neq 2 \wedge \delta_1 > 1].$$

We then create assertions which assert the equality of p_1 and p_2 with their corresponding expressions, and pass these along with Enc_ψ to the Solve procedure.

6 EVALUATION

To evaluate Plinko, we consider the following questions:

- (***Q1***) How efficiently can Plinko prove probabilistic properties on complex programs?
- (***Q2***) How does Plinko compare with other general purpose approaches?
- (***Q3***) What factors affect the performance of Plinko?
- (***Q4***) How do our optimizations impact the performance of Plinko?

We conducted our experiments on a machine with a 3.3GHz Intel Core i7-5820K and 32 GB of RAM, running Arch Linux with kernel 5.17.11.

(Q1) Discussion. We implemented each of our case studies presented in §4 in C++, and verified them using Plinko. We used publicly available implementations for the Bloom filter[2] and count-min sketch[3] case studies.

Like other symbolic execution methods, our tool requires concretizing some parameters, such as the sizes of input arrays and loop bounds. For instance, KLEE has limited support for unknown (symbolic) array lengths and so it instead considers concrete array sizes when exploring paths. Note that when an array length is concretized, the array contents remain symbolic. For example, if the input array length is concretized to 5 in quicksort, Plinko searches over all possible arrays of length 5. The choice of concretization also has a large effect on the performance of Plinko as larger concrete parameters generally produce more program paths, thus requiring more time to verify correctness. In a realistic verification setting where concrete parameters are not known ahead of time, Plinko could be used to consider longer and longer array inputs until it times out.

Table 1 summarizes our experimental results after verifying the target properties for each of the case studies presented in §4. For each experiment, we report the following metrics: the amount of time taken by Plinko to explore the paths of the program and to generate the query, the amount of time Z3 took to solve the query, the total amount of time elapsed, the number of lines in the C++ code, the number of paths Plinko explored, the number of random samples, and the maximum values of the concretized variables such that the query can be proven or disproven in the time allotted. (Recall that §4 describes the concretized parameters for each of our case studies.) All properties were verified within 10 minutes, except for Bloom filter due to a bug in the reference implementation. All results in Table 1 use the default version of Plinko (i.e., with only the simple guards optimization turned on).

We make a few general observations about the results. First, the case studies showcase a range of path counts and number of random samples. For example, even when we restricted the input array to be of length 5, Plinko still explored 120 paths containing 10 random samples when analyzing quicksort. Second, the time Plinko takes to execute Algorithm 1 is usually much less than the time

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

Table 1. Performance metrics for each of the case studies presented in §4. "Freivalds'" uses the SomeOff specification over ints and "Freivalds' (Multiple)" uses the FirstOff specification over chars.

	Timing (sec.)						
Case Study	**KLEE**	**Z3**	**Total**	**Lines**	**Paths**	**Samples**	**Concretizations**
Freivalds'	3	26	**29**	97	2	2	$n = 2$
Freivalds' (Multiple)	6	259	**265**	96	8	21	$(n, k) = (3, 7)$
Reservoir Sampling	14	98	**112**	52	127	6	$(n, k) = (13, 7)$
Reservoir Sampling	460	1	**461**	52	4096	12	$(n, k) = (13, 1)$
Monotone Testing	6	384	**390**	69	36	1	$n = 27$
Quicksort	14	114	**128**	65	120	10	$n = 5$
Bloom Filter	18	395	**413**	386	83	8	$(m, \varepsilon) = (3, 0.39)$
Count-min Sketch	4	145	**149**	245	3	8	$(n, \varepsilon, \gamma) = (4, 0.5, 0.25)$

Table 2. Performance metrics for each of the case studies presented in §4 under a 2-hour timeout. "Freivalds' (Multiple)" uses the FirstOff specification over chars.

	Timing (sec.)						
Case Study	**KLEE**	**Z3**	**Total**	**Lines**	**Paths**	**Samples**	**Concretizations**
Freivalds' (Multiple)	7	2027	**2034**	96	6	21	$(n, k) = (4, 5)$
Reservoir Sampling	5	85	**90**	52	15	3	$(n, k) = (41, 38)$
Reservoir Sampling	5	36	**41**	52	7	2	$(n, k) = (100, 98)$
Monotone Testing	6	5883	**5889**	69	42	1	$n = 32$

Z3 takes to solve the query. This suggests that the main bottleneck at the current scale of our case studies is constraint solving, not path exploration.

Extended Length Experiments. To better understand the scaling of Plinko, we ran additional experiments with a timeout of 2 hours. In Table 2 we report the performance results of these experiments. The concretization settings reported are the highest such that the target property can be successfully verified within 2 hours.

We first note that the concretization settings of Freivalds' algorithm, quicksort, Bloom filter, and count-min sketch could not be increased beyond those shown in Table 1. For Bloom filter we only tried to increase m and for count-min sketch we only tried to increase n. Additionally, only Freivalds' algorithm with $n = 3$ reached the 2-hour timeout, whereas quicksort, Bloom filter, and count-min sketch all ran out of memory before reaching the 2-hour timeout, at which point they were terminated by the operating system. For reservoir sampling, we show two concretized parameter settings with the highest resource usage (both time and memory). We note that the difficulty of the problem increases as n increases and as k decreases. When $(n, k) = (41, 38)$, we observed Plinko using a maximum of 17.8 GB of RAM, and when $(n, k) = (100, 98)$, Plinko peaked at 8.1 GB of RAM. On the other hand, Freivalds' (Multiple) and monotone testing were both more time constrained rather than memory constrained.

These extended experiments seem to suggest that memory usage is a large performance bottleneck for Plinko. Generally, Plinko consumes more memory as the number of paths and random samples increase. Memory consumption also increases as the cardinality of the originating distribution's

domain increases since we compute path probabilities by summing over every possible assignment to a random sample.

(Q2) Discussion. We compare Plinko against two state-of-the-art systems: Psi [Gehr et al. 2016, 2020], an exact symbolic inference engine for probabilistic programs, and Storm [Hensel et al. 2022], a leading probabilistic model checker used to check properties on finite-state probabilistic transition systems. We also discuss the differences between the input language of Plinko and those of Psi and Storm.

Psi Comparison. To compare against Psi we first implemented each of our case studies as probabilistic programs in the Psi language. Then, for each case study, we used Psi to produce a symbolic expression e_{PSI} which encodes the joint posterior distribution of the program parameterized by the input variables. We use e_{PSI} in place of Enc_ψ in our general formula for probabilistic queries: $\forall \alpha_1, \ldots, \alpha_n . e_{\text{PSI}} \bowtie f(\alpha_1, \ldots, \alpha_n)$, where $\alpha_1, \ldots, \alpha_n$ are universally quantified program inputs, and then use Z3 to solve the query.

However, we were only able to evaluate Psi on four of our case studies: Freivalds' (single and multiple), reservoir sampling, and monotonicity testing. For the Bloom filter and count-min sketch case studies, Psi returned an incorrect symbolic expression. Additionally, since Psi does not support recursive functions, we were unable to encode a comparable version of quicksort into Psi.

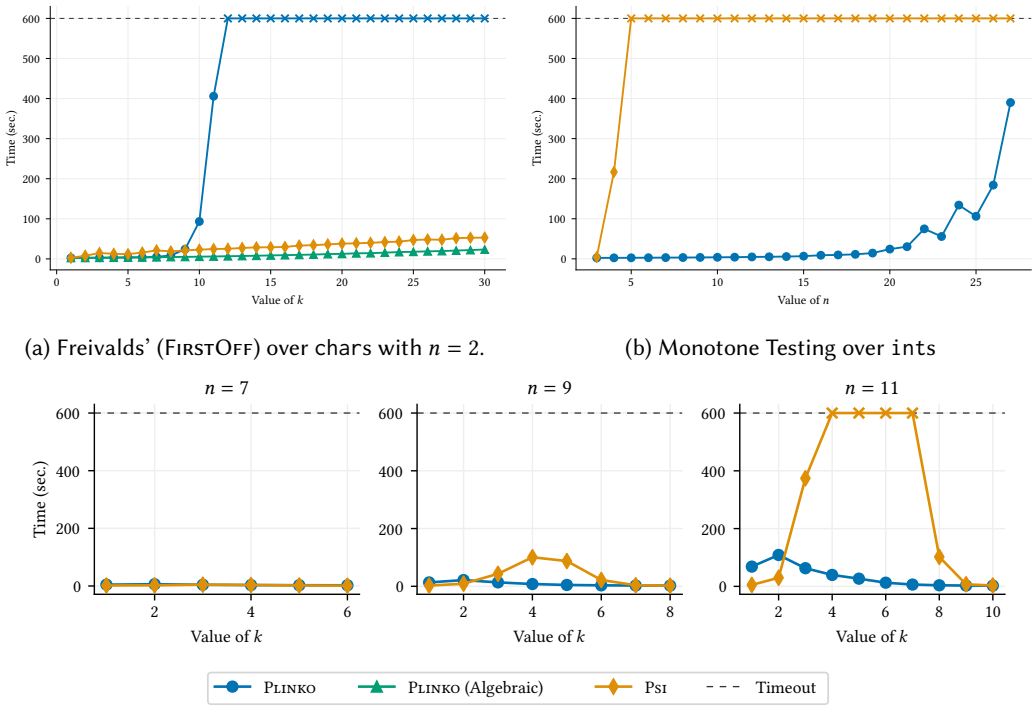
In Figure 6, we present performance results directly comparing Psi and Plinko on our first four case studies. We ran Psi and Plinko on our case studies with a range of concretized program parameters—like our system, Psi requires that some input parameters (e.g., sizes of input arrays) be concretized—and we report the end-to-end time for verification. For both tools, we used a 10-minute timeout denoted by a horizontal dashed line. An "X" means that for that experiment either a timeout was reached or the process ran out of memory. Recall that the default version of Plinko only employs the simple guards optimization.

Overall, we found that the performance of the default version of Plinko scales significantly better than Psi as the values of the concretized parameters increase. For monotone testing (Figure 6b) in particular, we found that Psi could only solve the query when $n = 3$ or 4, and timed out for all larger concretizations. For reservoir sampling, there are two parameters, n and k. Figure 6c shows how the tools perform as k varies, fixing $n = 7, 9$, and 11; we found similar results for other concrete values of n. For smaller values of n (e.g., $n = 7$), Psi and Plinko have about equal performance; however, as n increases we found that Plinko outperforms Psi. For instance, Plinko can handle all settings of k when $n = 11$ while Psi timed out for all $4 \leq k \leq 7$.

The only case study where the stock version of Plinko performs worse than Psi is Freivalds' algorithm (Figure 6a). While Psi significantly outperforms the default version of Plinko on this benchmark, if we enable algebraic simplifications optimizations, denoted by the green line in Figure 6a, Plinko was approximately 55% faster than Psi. We note that Psi performs similar algebraic simplifications.

Additionally, the vast majority of the time Plinko (Algebraic) spent was on executing KLEE instead of Z3. For this reason we hypothesize that the performance discrepancy between Psi and Plinko on Freivalds' algorithm is largely due to the relative ease KLEE has in exploring all paths, whereas Psi appears to be constrained by the path explosion caused by higher settings of n.

In summary, we found that Plinko is generally able to verify the evaluated benchmarks faster and for larger input concretizations than Psi. At the same time, we stress that Psi and Plinko are designed to solve different, yet complementary problems; the former performing exact probabilistic inference on probabilistic programs, and the latter verifying probabilistic properties over universally quantified input variables. One avenue for future work would be to explore how these two systems



(a) Freivalds' (FirstOff) over chars with $n = 2$.

(b) Monotone Testing over ints

(c) Reservoir Sampling where $n = 7$, 9, and 11.

Fig. 6. Performance results comparing Plinko against the Psi inference engine on the Freivalds' algorithm, Monotone Testing, and Reservoir Sample case studies. All experiments were run under a 10-minute timeout. An "X" marker means that an experiment either exceeded the timeout or ran out of memory.

could be merged such that Plinko leverages Psi's simplification engine to reduce the complexity and size of Plinko's symbolic expressions before calling Z3.

Storm Comparison. We compare the performance of Plinko and Storm on two of our case studies: Freivalds' algorithm and reservoir sampling. These two case studies involve reasoning about two different types of computations. Freivalds' algorithm mostly involves arithmetic operations (i.e., matrix-vector products), whereas reservoir sampling mainly copies and moves data.

While Plinko can operate on programs written in mainstream languages (e.g., C++), Storm operates on finite-state transition systems. Encoding our programs as transition systems is non-trivial. While our examples do have a finite state space, the state space is extremely large. Furthermore, a core aim of our work is to prove probabilistic properties that universally quantify over all possible program inputs, but Storm does not directly support programs with unknown inputs.

To work around these difficulties, we encoded our examples as Markov decision processes (MDPs) written in the PRISM format [Kwiatkowska et al. 2011]. To model a universal quantification over the program inputs, our encoded MDPs use a non-deterministic choice over all possible inputs. While the PRISM format made it easier to encode our imperative programs, the translation is not exact. For instance, the generated transition system does not accurately model finite precision arithmetic since the PRISM language does not support finite-width datatypes. Nevertheless, we believe that our encoding is a fair transition-system proxy for our target examples.

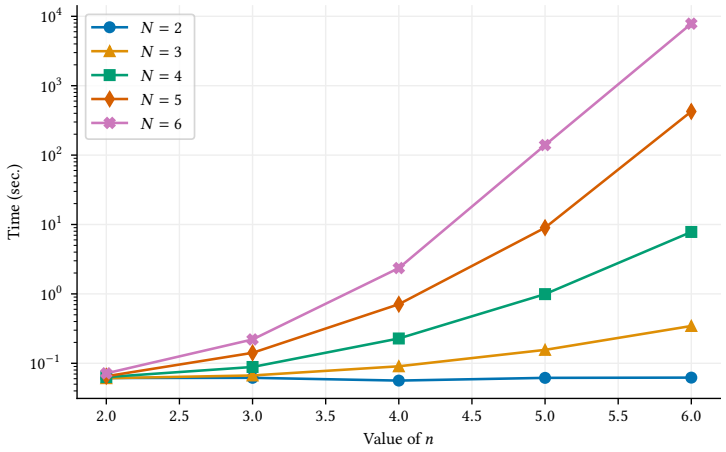


Fig. 7. Storm performance on Reservoir Sampling where $k = 1$, $n = 2..6$, and the array elements range from 1 to N. Note that the y-axis is on a *logarithmic* scale.

We discuss performance results, as well as the difference in guarantees that Storm and Plinko provide for both case studies below. For Freivalds' algorithm we solely consider 2×2 matrices. In our original C++ program, these matrices contain 32-bit integers. Due to the state space explosion, we needed to drastically reduce the domain size in order for Storm to successfully terminate. For example, when we allow the three matrices to only contain elements from the set $\{1, 2\}$, Storm can verify the upper bound for false positives in approximately 3 seconds where the constructed model contains roughly 42,000 states and 75,000 transitions. When we increased the domain to be $\{1, 2, 3\}$, however, we had to manually terminate the experiment after 14 hours had passed with no result. In contrast, Plinko is able to verify the property in 75 seconds where the input domain consists of all 32-bit integers.

To rule out the chance that our encoding of Freivalds' algorithm as a non-deterministic choice over all possible inputs was not the bottleneck, we also ran Storm on each possible concrete setting of the three 2×2 input matrices over the domains $\{1, 2\}$ and $\{1, 2, 3\}$. For the domain $\{1, 2\}$, this method took substantially more time, finishing in just over 20 seconds, while the domain $\{1, 2, 3\}$ also took more than 12 hours before we manually terminated the experiment.

We saw similar behavior with reservoir sampling. To understand the dependence on the data size, we considered five input domains $\{1, \ldots, N\}$, where $2 \leq N \leq 6$. For each input domain, we ran Storm on five different settings of n and k, where $k = 1$ and $2 \leq n \leq 6$. Figure 7 presents the performance results from these experiments. Note that as the domain size increases linearly, the time required to check the property increases *exponentially*. As we will show in (***Q3***), Plinko also sees decreased performance as we increase the width of program variables, and we expect to see similar exponential scaling if we continued increasing the size of variables. However, Plinko is able to check the property for all settings of n in approximately 3 seconds when the input domain consists of all 32-bit integers.

Input Language Differences. We conclude our evaluation against Psi and Storm by discussing the source language differences between these three tools. As stated in §5, Plinko consumes LLVM bytecode which allows us to verify programs written in any language supported by the LLVM front-end (e.g., C, C++, Objective C, Fortran, etc.). This broadens the applicability of our analysis as most mainstream programming languages are able to compile down to LLVM; however, this also significantly increases the verification complexity as LLVM IR uses more complex datatypes, such

Table 3. Performance metrics of three different ways of specifying $A \times B \neq C$ in the query for Freivalds' algorithm. Here, $n = 2$ and all elements of the matrices are C++ `int`s.

	Timing (sec.)			
Spec. for $A \times B \neq C$	**KLEE**	**Z3**	**Total**	**Paths**
AllOff	3	1	**4**	2
SomeOff	3	26	**29**	2
FirstOff	2	5	**7**	2

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

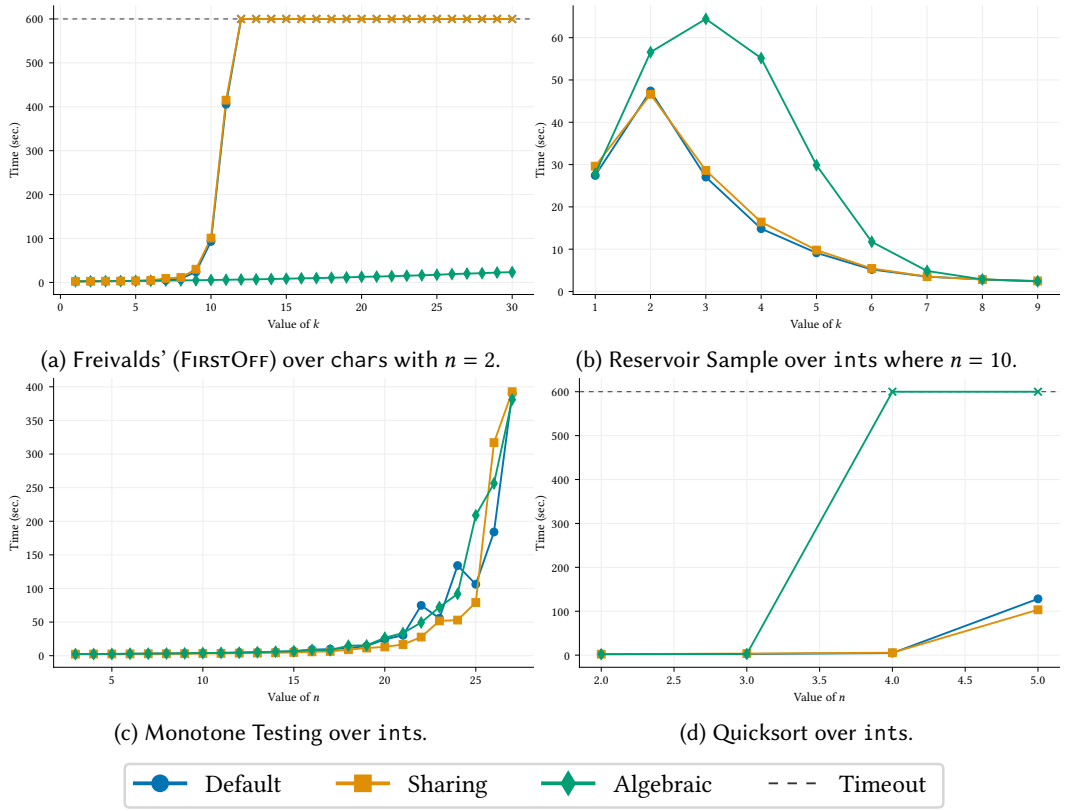
	Timing (sec.)		
Data Type	**KLEE**	**Z3**	**Total**
`long int`	4	684	**688**
`int`	3	26	**29**
`short int`	2	4	**6**
`char`	2	1	**3**

as machine integers, arrays, etc. In particular, Plinko natively reasons over fixed-width bitvectors, as opposed to infinite precision, mathematical integers, allowing Plinko to detect bugs related to integer overflow/underflow.

Psi and Storm, on the other hand, operate on more idealized core languages. Psi provides a custom domain-specific language which does not support as many programming features as mainstream languages, such as recursion and fixed-width integers, and can only be used in conjunction with Psi (the programs themselves cannot be run, only analyzed by Psi). Storm, as a probabilistic model checker, requires the programmer to encode their programs as either a discrete- or continuous-time Markov model instead of executable LLVM code. For our evaluation, we use Markov decision processes (MDPs), which exactly encodes a probabilistic program with (a) a bounded, finite space of inputs and (b) exact arithmetic with *infinite* precision. While this is not an exact match for the semantics of our programming language, it is unclear how to generate MDPs that match the semantics of LLVM bytecode. In any case, we do not believe that increasing the realism of the MDP encoding would significantly improve the performance of Storm.

(Q3) Discussion. Recall that Freivalds' algorithm efficiently determines whether $A \times B = C$, where A, B, and C are $n \times n$ matrices; however, the algorithm has a false positive error rate of at most $1/2$ if $A \times B \neq C$. In order to verify this error rate, we must first assume that $A \times B \neq C$, and we propose three different encodings: AllOff, SomeOff, and FirstOff. To determine the performance impact for each specification we ran Freivalds' algorithm where $k = 1$ with 2×2 matrices which each contain 32-bit integers. We present these performance results in Table 3.

In general, the results in Table 3 suggest that simpler and more specific encodings of $A \times B \neq C$ increase performance at the cost of missing potential bugs. The strongest and best performing specification was AllOff, whereas the most general and worst performing specification was SomeOff. Intuitively, this makes sense as AllOff restricts the search space considerably more than the other specifications, whereas SomeOff requires Z3 to reason about *all* matrices A, B, and



(a) Freivalds' (FirstOff) over chars with $n = 2$.

(b) Reservoir Sample over ints where $n = 10$.

(c) Monotone Testing over ints.

(d) Quicksort over ints.

Fig. 8. Performance metrics comparing the default, unoptimized version of Plinko, Plinko with the formula sharing ("Sharing"), and Plinko with algebraic simplifications ("Algebraic"). An "X" means that the experiment either exceeded the 10-minute timeout, or ran out of memory.

C such that $A \times B \neq C$. Therefore, while performing the worst, SomeOff provides the strongest guarantee out of all the other variants, followed by FirstOff, and then finally, AllOff.

We also consider how the size of the domain (i.e., the C++ datatype) of the matrix elements impacts performance. If each element of the matrix is only a single byte, there are only 2^{32} possible 2×2 matrices, as opposed to elements of eight bytes, of which there are 2^{256} possible matrices. For each data type, we again restrict ourselves to 2×2 matrices and specify that $A \times B \neq C$ using the SomeOff encoding. The performance results are presented in Table 4. On the evaluating machine, `long ints` are eight bytes, `ints` are four bytes, `short ints` are two bytes, and `chars` are a single byte.

Unsurprisingly, we found a direct correspondence between the size of the integer and the time it took to verify the property. The time it took to verify Freivalds' algorithm with each data type seems to increase exponentially with the corresponding increase in integer size. We must note, however, that using `chars` provides the weakest guarantee whereas `long ints` provides the strongest. Although, for many applications, analyzing variants of the program with smaller data types might already provide sufficient confidence of correctness, or may surface bugs.

(Q4) Discussion. We consider the performance impacts of our three main optimizations: precomputing the probability of taking branches containing simple guards, applying algebraic simplifications, and sharing formulas to exploit any overlap between paths.

The simple guard optimization had a large effect on our two case studies with simple guards: reservoir sampling and monotone testing. Without this optimization, we could only verify the target properties for reservoir sampling when $(n, k) = (10, 5)$ and monotone testing when $n = 23$ in 128 seconds and 356 seconds, respectively. Increasing the concretized values any higher caused solving to either exceed the 10-minute timeout, or to run out of memory. Comparatively, we are able to explore larger concretizations under the same time limit ($(n, k) = (13, 7)$, and $n = 27$) with this optimization. These results show that it is possible to compute path probabilities more compactly, and suggest that further optimizations in this style may be possible.

To evaluate the effectiveness of the remaining two optimizations, we ran Plinko on the first four case studies (we excluded the Bloom filter and count-min sketch case studies due to the number of concretized variables) with the default version of Plinko, Plinko with the algebraic simplifications optimization ("Algebraic"), and Plinko with the formula sharing optimization ("Sharing"). Performance results for each of the case studies are summarized in Figure 8.

The results show that these two optimizations are not always effective on all programs, however, they can have a significant impact on performance, as showcased by the algebraic simplifications optimization on Freivalds' algorithm (Figure 8a). At the same time, we see that this optimization's impact is not always positive, as is the case with reservoir sampling and quicksort. This is not so surprising as performance gains from this optimization can only occur in programs with large amounts of arithmetic, e.g., Freivalds' algorithm in the form of matrix-vector products. The other case studies involve more comparisons and memory manipulations so the overhead of enabling this optimization is present without any of the benefits. The sharing optimization increases performance slightly on the monotone testing case study, but otherwise saw around equal performance when compared to the stock version of Plinko.

In general, Figure 8 suggests that optimizations can significantly improve performance, but they are ultimately heuristics. Accordingly, we envision that in order to get the most out of the optimizations, Plinko should be run in parallel with different combinations enabled. We hope to both refine and create more optimizations in future work.

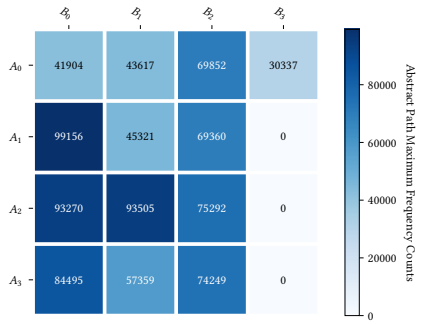
7 FURTHER POSSIBLE OPTIMIZATIONS: TOWARDS PATH FILTERING

In our evaluation, (***Q4***) shows how simplifying the query generated by Plinko can lead to significant performance gains. In this section, we consider a different route for optimizations: path filtering. We ask: of the paths explored by Algorithm 1, is there a *small* subset of paths that contain most of the probability mass? Clearly, we cannot hope for this to be so for all probabilistic programs; however, our experiments do suggest that such a subset does exist for some realistic probabilistic programs. This does not immediately yield an optimization for Algorithm 1—it is not clear how to prune paths automatically. Still, it does indicate that a path pruning heuristic could further improve the performance of Algorithm 1.

Path partitioning. Here we develop a path filtering scheme which aims to remove low-probability paths by selecting a subset of paths which account for the majority of the target probability mass. This leads to smaller formulas that are hopefully faster to solve. While the goal is simple, there are a few technical challenges. First, the probability of a path may depend on the setting of the input variables—the same path may have high probability for some inputs, and low probability for other inputs. Second, the space of program inputs can be very large, potentially even infinite.

```
1:  function CoinFlips(b1, b2)
2:    heads ← 0
3:    for i ← 1, 3 do
4:      t1 ~ UniformInt(0, 5)
5:      t2 ~ UniformInt(0, 5)
6:      if t1 ≥ b1 then
7:        heads ← heads + 1
8:      if t2 ≥ b2 then
9:        heads ← heads + 1
10:   return heads
```

(a) The CoinFlips program.



(b) Frequency count heatmap.

Fig. 9. The CoinFlips program along with a heatmap showing the frequency counts for each of the 16 abstract paths defined by Γ.

To address these issues, we first partition all paths into a bounded set of *abstract paths* using a user-provided partitioning function, $\Gamma : A_\forall \times A_p \to \mathbb{N}$, where $A_\forall$ is the set of all assignments of universal symbolic variables (i.e., program inputs) to values, and A_p is the set of all assignments of probabilistic symbolic variables to values. Note that a pair of assignment functions $(a_\forall, a_p) \in A_\forall \times A_p$ fully determine which path a program will follow. So, we define $\Gamma(a_\forall, a_p) = n$ to mean that the program path described by $a_\forall$ and a_p belongs to the abstract path n.

Referencing program paths using $a_\forall$ and a_p is not a one-to-one mapping, however, as it is quite possible that there exist alternative assignment functions, $a'_\forall$ and a'_p that produce the same path as $a_\forall$ and a_p. For this reason we require Γ to be *path disjoint*: if $(a_\forall, a_p)$ and $(a'_\forall, a'_p)$ produce the same path, then $\Gamma(a_\forall, a_p) = \Gamma(a'_\forall, a'_p)$. Additionally, Γ should ideally induce an *asymmetric distribution*, meaning that Γ should partition the abstract paths such that most of the target probability mass is concentrated in a few abstract paths, allowing us to prune away a greater number of low-probability paths.

Now that we have partitioned all paths into a number of abstract paths, we need to determine which abstract paths to filter out. We do this by estimating which abstract paths have high-probability mass using statistical sampling and counting. For each abstract path n we randomly sample $k_\forall$ assignment functions of the input variables $a^1_\forall, \ldots, a^{k_\forall}_\forall$ such that there exists an a_p where $\Gamma(a^i_\forall, a_p) = n$. In other words, we randomly sample settings of the program inputs such that it is possible to classify the resulting path as a member of the abstract path n. Then, for each $a^i_\forall$, we apply the substitution $a^i_\forall$ on the input variables, execute the program k_p times, and record the resulting assignments of the probabilistic variables, namely $a^1_p, \ldots, a^{k_p}_p$. We then count how many of the produced probabilistic assignments result in the abstract path n. In other words, we compute $C(a^i_\forall) \triangleq \sum_{j=1}^{k_p} [\Gamma(a^i_\forall, a^j_p)]$, where $[\cdot]$ are Iverson brackets. We estimate how often the abstract path n occurs by calculating $\max_{1 \le i \le k_\forall} C(a^i_\forall)/k_p$. Lastly, we prune all abstract paths that have a probability of occurring less than some user-defined threshold.

Example. To illustrate, consider the program in Figure 9a. This program generates samples from two biased coin-flip distributions three times each and returns the total number of heads. The program inputs $b_1, b_2 \in [0, 5]$ determine the bias of the two biased coins. We would like to verify that the *maximum* expected value of *heads* is 6 (i.e., when $b_0 = b_1 = 0$, or both coins always return heads).

First, we have to define our partitioning function Γ. Let $A \in \{A_0, \ldots, A_3\}$ be a set of input variable classes, $B = \{B_0, \ldots, B_3\}$ be a set of probabilistic variable classes, and t^i_1 and t^i_2 be the

corresponding random samples of iteration i of Figure 9a, where $1 \leq i \leq 3$. Each input variable class corresponds to a set of assignments to the input variables, and each probabilistic variable class similarly corresponds to a set of assignment to the probabilistic variables. In particular,

$$A_0 \triangleq b_1 > t_1^0 \wedge b_2 > t_2^0 \qquad B_0 \triangleq S \geq 4 \wedge S \leq 6$$
$$A_1 \triangleq b_1 \leq t_1^0 \wedge b_2 \leq t_2^0 \qquad B_1 \triangleq S = 3$$
$$A_2 \triangleq b_1 \leq t_1^0 \wedge b_2 > t_2^0 \qquad B_2 \triangleq S = 2 \vee S = 1$$
$$A_3 \triangleq b_1 > t_1^0 \wedge b_2 \leq t_2^0 \qquad B_3 \triangleq S = 0$$

where $S \triangleq \sum_{i=1,j=1}^{i=2,j=3} g_{ij}$ and $g_{ij} = 0$ if $t_1^j \geq b_i$ and 1 otherwise. We then define an abstract path to be a pair of input variable and probabilistic variable classes $(A_i, B_j) \in A \times B$. Therefore, $\Gamma(a_\forall, a_p) = (A_i, B_j)$ if $a_\forall \in A_i$ and $a_p \in B_j$.

We show the maximum frequency counts for each of the 16 abstract paths as a heatmap in Figure 9b. Note that many of the abstract paths hold little probability mass, particularly those defined using B_3. Using path filtering, we were able to verify that the maximum expected number of heads is indeed 6 using just 6 of the 16 abstract paths defined by Γ. We stress that we are able to eliminate the remaining 10 abstract paths because Γ induces an asymmetric distribution on the abstract paths. An alternative partitioning function Γ' which did not induce an asymmetric distribution required 12 abstract paths to verify the same property.

Experimental results. We invoked path filtering with hand-crafted partitioning functions on our case studies, and measured the percentage speedup, path reduction, and error compared to baseline Algorithm 1. To measure error, we found the minimum additive bound ε such that for all inputs the probability expression Enc_ψ is off by at most ε from the target bound. Some experiments saw large speedups (30.2% for Bloom filter, 96.0% speedup for monotone testing) and significant decreases in the number of paths (32.5% for Bloom filter to 50.4% for reservoir sampling), while incurring a small additive error bound ε (ranging from 0.0 to 0.18). These results suggest that given a sufficiently good partitioning scheme, path filtering can produce significant performance gains for Algorithm 1. The main challenge is finding the partitioning automatically, which we leave for future work.

8 RELATED WORK

Probabilistic Symbolic Execution. Geldenhuys et al. [2012] first proposed a method for probabilistic symbolic execution. Given a standard, non-probabilistic program (i.e., programs without random sampling statements), their technique assumes all inputs are drawn from a discrete uniform distribution, and then computes the probabilities of program paths using model counting. Their tool then produces the posterior distribution parameterized by the return values of the programs which they use for bug finding and testing purposes.

While our technique and theirs both use a form of probabilistic symbolic variables and symbolic execution, there are a few crucial differences. First, Plinko operates on randomized programs with random sampling statements throughout the program instead of standard, deterministic programs with randomized inputs. Second, Plinko treats input variables as universally quantified, rather than assuming that inputs are uniformly distributed. Just like typical properties of deterministic programs, target properties of randomized programs usually quantify over all input variables. Supporting universally-quantified inputs makes symbolic execution more challenging. While branch probabilities in Geldenhuys et al. [2012]'s setting are numeric constants, which allow Geldenhuys et al. [2012] to leverage methods like model counting and volume estimation, branch probabilities in our setting are symbolic expressions that can mention program inputs. Accordingly, reasoning about path probabilities is more difficult in our setting. Since all of our benchmarks aim to verify

properties in the presence of unknown inputs, they cannot be handled using existing probabilistic symbolic execution methods. Finally, Plinko's main application is to verify probabilistic properties (e.g., false positive rates, expected value bounds) whereas Geldenhuys et al. [2012] uses probabilistic symbolic execution as a bug-finding tool by automatically calculating, but manually analyzing, path probabilities.

Later works use this idea for different applications: analyzing software reliability [Borges et al. 2014; Filieri et al. 2013], quantifying software changes [Filieri et al. 2015], generating performance distribution [Chen et al. 2016], and evaluating worst-case input distributions [Kang et al. 2021]. Recent schemes apply volume computation instead of model counting, which is a performance bottleneck [Albarghouthi et al. 2017; Sankaranarayanan et al. 2013].

Existing methods work with probabilistic programs where program inputs are either known constants, or sampled from known distributions (often, the uniform distribution). In contrast, our technique quantifies over all unknown inputs, rather than assuming they are drawn from fixed distributions.

Symbolic inference. Probabilistic programming languages (PPLs) are languages enriched with both *sampling* and *conditioning* operations. These two features allow probabilistic programs to encode complex distributions. In fact, many models of interest in machine learning can be expressed in this way. A basic task is *inference*: given an assertion P, what is the probability that P holds in the distribution described by the program? Researchers have considered a variety of approaches, from weighted model counting [Holtzen et al. 2020], to analyzing Bayesian networks [Sampson et al. 2014], to applying computer algebra systems [Claret et al. 2013; Gehr et al. 2016, 2020].

Most existing probabilistic programming languages assume that inputs are drawn from known distributions—this is a natural simplification since PPLs are typically concerned with analyzing a single complex distribution, rather than a family of distributions—so they cannot be applied to prove our properties of interest. Some recent PPLs do support reasoning about programs with unknown parameters. Probably the most relevant such system is Psi [Gehr et al. 2016, 2020]. Like Plinko, Psi is designed for answering exact, symbolic queries about distributions generated by probabilistic programs. Furthermore, Psi supports programs with unknown parameters, much like our target programs. This is not a typical use-case of Psi—these features aren't documented in the main paper, and we encountered cases where Psi failed to compute the correct probabilities—but Psi is capable of analyzing some of our benchmarks. As our evaluation in §6 shows, Plinko enjoys significantly better performance and scaling when verifying probabilistic programs.

Overall, the comparison with PPLs is imperfect because PPLs are optimized for reasoning about programs which use *conditioning*, an operation that is not supported by Plinko. While conditioning is rarely used as an operation in randomized algorithms—precisely because of its computational intractability—it would be interesting to extend our work to handle conditioning.

Other automated methods for probabilistic programs. Automated verification of probabilistic programs is an active and diverse area of research; we briefly survey several main lines of work.

AxProf [Joshi et al. 2019] uses *statistical testing* to analyze probabilistic programs, by running the target program multiple times on concrete inputs in order to estimate probabilities and expected values. AxProf is highly efficient and supports programs with unknown inputs. Unlike our work, however, it can only explore a small subset of the input space and cannot provide logical guarantees.

Probabilistic model checking is a well-developed method for checking logical formulas on probabilistic transition systems [Baier et al. 1997, 2018; Hensel et al. 2022; Kwiatkowska et al. 2011]. These techniques can check properties that are not easily handled by probabilistic symbolic execution, since assertions can be written in a variety of temporal logics. However, our evaluation (cf. (***Q2***))

shows that probabilistic model checkers perform much worse than our approach on our target programs and properties.

Abstract interpretation and *algebraic program analysis* methods have been developed for probabilistic programs [Cousot and Monerau 2012; Wang et al. 2018]. These methods abstract the probabilistic state, trading precision in exchange for tractable analysis. Our method computes path probabilities exactly in a symbolic form. It would be interesting to see if we can leverage abstract interpretation ideas to avoid concretizing loop bounds and unrolling loops.

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

Deductive verification for probabilistic programs. Finally, there is a wide variety of manual and semi-automated methods for verifying probabilistic programs, which we cannot hope to fully survey here. Perhaps the most well-developed method is Morgan and McIver's weakest pre-expectation calculus [Gretz et al. 2014; Morgan et al. 1996], which manipulates quantitative assertions for probabilistic programs [Kozen 1985]. In contrast to our method, this is not easy to automate (though there have been some efforts [Bao et al. 2022; Gretz et al. 2013]), and targets the core probabilistic language pGCL rather than a real implementation language. Interested readers can consult the recent monograph [Barthe et al. 2020] for an overview of other methods.

9 CONCLUSION AND FUTURE DIRECTIONS

We have presented a symbolic execution method for randomized programs to in order to automatically verify probabilistic properties which quantify over all unknown inputs. Going forward, we see at least two promising directions for further investigation.

Optimizing probabilistic symbolic execution. In this work, we have made only preliminary efforts to optimize our symbolic execution method. One natural direction is to develop heuristics for exploring paths. Our experiments in §7 suggest that path filtering can reduce the number of paths that must be explored. Another possibility is to develop better methods for simplifying path probability expressions, along the lines of the three optimizations presented in §5.

Analyzing more complex probabilistic programs. So far, we have evaluated our implementation on standard randomized programs. Both KLEE and Z3 support richer programs and hardware features. For instance, Z3 has support for reasoning about floating-point arithmetic. Recent work develops an extension of KLEE that works on unbounded integers [Kapus et al. 2019]; it could be interesting to see if this technique has better performance when verifying randomized algorithms, which often work with mathematical integers.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their detailed feedback. This work benefited from discussions with Dexter Kozen, Adrian Sampson, and Cornell PLDG. This work was partially supported by the National Science Foundation (Grant No. 1943130), the University of Wisconsin–Madison, and Cornell University. The second author is fully supported and the fourth author is partially supported by TCS Research via the TCS Research Scholar Fellowship program.

REFERENCES

Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. 2017. FairSquare: Probabilistic Verification of Program Fairness. *Proceedings of the ACM on Programming Languages* 1, OOPSLA, Article 80 (2017). https://doi.org/10.1145/3133904

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

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

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

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

Jialu Bao, Nitesh Trivedi, Drashti Pathak, Justin Hsu, and Subhajit Roy. 2022. Data-Driven Invariant Learning for Probabilistic Programs. In *International Conference on Computer Aided Verification (CAV), Haifa, Israel*. https://doi.org/10.1007/978-3-031-13185-1_3 arXiv:2106.05421

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

Gilles Barthe, Rohit Chadha, Paul Krogmeier, A. Prasad Sistla, and Mahesh Viswanathan. 2021. Deciding Accuracy of Differential Privacy Schemes. *Proceedings of the ACM on Programming Languages* 5, POPL, Article 8 (Jan. 2021). https://doi.org/10.1145/3434289

Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva (Eds.). 2020. *Foundations of Probabilistic Programming Languages*. Cambridge University Press. 145–184 pages. https://doi.org/10.1017/9781108770750.006

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

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

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

Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

Mateus Borges, Antonio Filieri, Marcelo d'Amorim, Corina S. Păsăreanu, and Willem Visser. 2014. Compositional Solution Space Quantification for Probabilistic Software Analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Edinburgh, Scotland*. 123–132. https://doi.org/10.1145/2666356.2594329

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Diego, California*. 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C Rinard. 2012. Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Beijing, China*. 169–180. https://doi.org/10.1145/2254064.2254086

Aleksandar Chakarov and Sriram Sankaranarayanan. 2013. Probabilistic Program Analysis with Martingales. In *International Conference on Computer Aided Verification (CAV), Saint Petersburg, Russia (Lecture Notes in Computer Science, Vol. 8044)*. 511–526. https://doi.org/10.1007/978-3-642-39799-8_34

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

Bihuan Chen, Yang Liu, and Wei Le. 2016. Generating Performance Distributions via Probabilistic Symbolic Execution. In *International Conference on Software Engineering (ICSE), Austin, Texas*. 49–60. https://doi.org/10.1145/2884781.2884794

Guillaume Claret, Sriram K. Rajamani, Aditya V. Nori, Andrew D. Gordon, and Johannes Borgström. 2013. Bayesian Inference Using Data Flow Analysis. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), Saint Petersburg, Russia*. 92–102. https://doi.org/10.1145/2491411.2491423

Graham Cormode and S. Muthukrishnan. 2004. An Improved Data Stream Summary: The Count-Min Sketch and Its Applications. In *Latin American Symposium on Theoretical Informatics (LATIN), Buenos Aires, Argentina (Lecture Notes in Computer Science, Vol. 2976)*. Springer-Verlag, 29–38. https://doi.org/10.1007/978-3-540-24698-5_7

Patrick Cousot and Michael Monerau. 2012. Probabilistic Abstract Interpretation. In *European Symposium on Programming (ESOP), Tallinn, Estonia (Lecture Notes in Computer Science, Vol. 7211)*. Springer-Verlag, 169–193. https://doi.org/10.1007/978-3-642-28869-2_9

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary (Lecture Notes in Computer Science, Vol. 4963)*. Springer-Verlag, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. 2019. Relational Symbolic Execution. In *ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP), Porto, Portugal*. Article 10. https://doi.org/10.1145/3354166.3354175

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

Antonio Filieri, Corina S. Păsăreanu, and Willem Visser. 2013. Reliability Analysis in Symbolic Pathfinder. In *International Conference on Software Engineering (ICSE), San Francisco, California*. 622–631. https://doi.org/10.1109/ICSE.2013.6606608

Antonio Filieri, Corina S. Păsăreanu, and Guowei Yang. 2015. Quantification of Software Changes through Probabilistic Symbolic Execution. In *IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, Nebraska*. 703–708. https://doi.org/10.1109/ASE.2015.78

Rūsiņš Freivalds. 1977. Probabilistic Machines Can Use Less Running Time. In *IFIP World Congress, Toronto, Canada*. North-Holland, 839–842.

Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. In *International Conference on Computer Aided Verification (CAV), Toronto, Ontario (Lecture Notes in Computer Science, Vol. 9779)*. Springer-Verlag, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

Timon Gehr, Samuel Steffen, and Martin Vechev. 2020. λPSI: Exact Inference for Higher-Order Probabilistic Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), London, England*. 883–897. https://doi.org/10.1145/3385412.3386006

Jaco Geldenhuys, Matthew B. Dwyer, and Willem Visser. 2012. Probabilistic Symbolic Execution. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Minneapolis, Minnesota*. 166–176. https://doi.org/10.1145/2338965.2336773

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

Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. 2013. Prinsys—On a Quest for Probabilistic Loop Invariants. In *International Conference on Quantitative Evaluation of Systems (QEST), Buenos Aires, Argentina (Lecture Notes in Computer Science, Vol. 8054)*. Springer-Verlag, 193–208. https://doi.org/10.1007/978-3-642-40196-1_17

Friedrich Gretz, Joost-Pieter Katoen, and Annabelle McIver. 2014. Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. *Performance Evaluation* 73 (2014), 110–132. https://doi.org/10.1016/j.peva.2013.11.004

Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2022. The probabilistic model checker Storm. *International Journal on Software Tools for Technology Transfer* 24, 4 (2022), 589–610. https://doi.org/10.1007/s10009-021-00633-z arXiv:2002.07080

C. A. R. Hoare. 1961. Algorithms 63-64: partition and quicksort. *Commun. ACM* 4, 7 (July 1961), 321. https://doi.org/10.1145/366622.366642

Steven Holtzen, Guy Van den Broeck, and Todd Millstein. 2020. Scaling Exact Inference for Discrete Probabilistic Programs. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 140 (Nov. 2020). https://doi.org/10.1145/3428208

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

Qiao Kang, Jiarong Xing, Yiming Qiu, and Ang Chen. 2021. Probabilistic Profiling of Stateful Data Planes for Adversarial Testing. In *International Conference on Architectural Support for Programming Langauages and Operating Systems (ASPLOS)*. 286–301. https://doi.org/10.1145/3445814.3446764

Timotej Kapus, Martin Nowack, and Cristian Cadar. 2019. Constraints in Dynamic Symbolic Execution: Bitvectors or Integers?. In *International Conference on Tests and Proofs (TAP), Porto, Portugal (Lecture Notes in Computer Science, Vol. 11823)*. Springer-Verlag, 41–54. https://doi.org/10.1007/978-3-030-31157-5_3

Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. 2012. HAMPI: A Solver for Word Equations over Strings, Regular Expressions, and Context-Free Grammars. *ACM Transactions on Software Engineering and Methodology* 21, 4, Article 25 (Nov. 2012). https://doi.org/10.1145/2377656.2377662

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

Dexter Kozen. 1985. A Probabilistic PDL. *J. Comput. System Sci.* 30, 2 (1985), 162–178. https://doi.org/10.1016/0022-0000(85)90012-1

Marta Kwiatkowska, Gethin Norman, and David Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *International Conference on Computer Aided Verification (CAV), Snowbird, Utah (Lecture Notes in Computer Science, Vol. 6806)*. Springer-Verlag, 585–591. https://doi.org/10.1007/978-3-642-22110-1_47

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

Carroll Morgan, Annabelle McIver, and Karen Seidel. 1996. Probabilistic Predicate Transformers. *ACM Transactions on Programming Languages and Systems* 18, 3 (1996), 325–353. https://doi.org/10.1145/229542.229547

Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. 2014. Expressing and Verifying Probabilistic Assertions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Edinburgh, Scotland*. 112–122. https://doi.org/10.1145/2594291.2594294

Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. 2013. Static Analysis for Probabilistic Programs: Inferring Whole Program Properties from Finitely Many Paths. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Seattle, Washington*. 447–458. https://doi.org/10.1145/2499370.2462179

Raimondas Sasnauskas, Oscar Soria Dustmann, Benjamin Lucien Kaminski, Klaus Wehrle, Carsten Weise, and Stefan Kowalewski. 2011. Scalable Symbolic Execution of Distributed Systems. In *International Conference on Distributed Computing Systems (ICDCS), Minneapolis, Minnesota*. 333–342. https://doi.org/10.1109/ICDCS.2011.28

Raimondas Sasnauskas, Olaf Landsiedel, Muhammad Hamad Alizai, Carsten Weise, Stefan Kowalewski, and Klaus Wehrle. 2010. KleeNet: Discovering Insidious Interaction Bugs in Wireless Sensor Networks before Deployment. In *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), Stockholm, Sweden*. 186–196. https://doi.org/10.1145/1791212.1791235

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

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

Zachary Susag, Sumit Lahiri, Justin Hsu, and Subhajit Roy. 2022. Artifact for Symbolic Execution for Randomized Programs. https://doi.org/10.5281/zenodo.7061819

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

Di Wang, Jan Hoffmann, and Thomas W. Reps. 2018. PMAF: An Algebraic Framework for Static Analysis of Probabilistic Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadelphia, Pennsylvania*. 513–528. https://doi.org/10.1145/3192366.3192408

Di Wang, Jan Hoffmann, and Thomas W. Reps. 2021. Central Moment Analysis for Cost Accumulators in Probabilistic Programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 559–573. https://doi.org/10.1145/3453483.3454062 arXiv:2001.10150