Expressing and Verifying Probabilistic Assertions

Adrian Sampson Pavel Panchekha Todd Mytkowicz Kathryn S. McKinley Dan Grossman Luis Ceze
University of Washington Microsoft Research University of Washington

Abstract

Traditional assertions express correctness properties that must hold on every program execution. However, many applications have probabilistic outcomes and consequently their correctness properties are also probabilistic (e.g., they identify faces in images, consume sensor data, or run on unreliable hardware). Traditional assertions do not capture these correctness properties. This paper proposes that programmers express probabilistic correctness properties with *probabilistic assertions* and describes a new *probabilistic evaluation* approach to efficiently verify these assertions. Probabilistic assertions are Boolean expressions that express the *probability* that a property will be true in a given execution rather than asserting that the property must always be true. Given either specific inputs or distributions on the input space, probabilistic evaluation verifies probabilistic assertions by first performing *distribution extraction* to represent the program as a Bayesian network. Probabilistic evaluation then uses statistical properties to simplify this representation to efficiently compute assertion probabilities directly or with sampling. Our approach is a mix of both static and dynamic analysis: distribution extraction statically builds and optimizes the Bayesian network representation and sampling dynamically interprets this representation. We implement our approach in a tool called MAYHAP for C and C++ programs. We evaluate expressiveness, correctness, and performance of MAYHAP on programs that use sensors, perform approximate computation, and obfuscate data for privacy. Our case studies demonstrate that probabilistic assertions describe useful correctness properties and that MAYHAP efficiently verifies them.

Categories and Subject Descriptors G.3 [*Probability and Statistics*]: Statistical computing; D.2.5 [*Software Engineering*]: Testing and Debugging—Symbolic execution

General Terms Languages, Reliability

Keywords Probabilistic programming, approximate computing, data obfuscation, differential privacy, sensors, symbolic execution

1. Introduction

Traditional assertions express logical properties that help programmers design and reason about the correctness of their program. Verification tools guarantee that every execution will satisfy an assertion, such as the absence of null dereferences or a legal value range for a variable. However, many applications produce or consume probabilistic data, such as the relevance of a document to a search, the distance to the nearest coffee shop, or the estimated arrival time of the next bus. From smartphones with sensors to robots to machine learning to big data to approximate computation, many applications compute with probabilistic values.

Current assertion languages and verification tools are insufficient in this domain. Traditional assertions do not capture probabilistic correctness because they demand that a property hold on *every* execution. Recent work on inference in probabilistic programming languages builds language abstractions to aid programmers in describing machine learning models but does not deal with verification of probabilistic correctness properties [13, 22, 26, 28]. Sankaranarayanan et al. [33] address the verification of programs in probabilistic programming languages through polyhedral volume estimation, but this approach limits the domain to programs with linear arithmetic over constrained probability distributions. In contrast, this paper builds on Bornholt et al.'s $Uncertain\langle T\rangle$ [3], which defines a semantics for computing in mainstream languages over a broader set of distributions with sampling functions but does not verify programs.

This paper proposes *probabilistic assertions* (passerts), which express probabilistic program properties, and *probabilistic evaluation*, which verifies them. A passert statement is a probabilistic logical statement over random variables. *Probabilistic evaluation* extracts, optimizes, and evaluates the distribution specified in a passert by combining techniques from static verification, symbolic execution, and dynamic testing.

Probabilistic Assertions Programmers write **passert** e, p, cf to check the probability that the Boolean expression e holds in a given execution of the program is at least p with confidence cf. The parameters p (defaults to 0.5) and cf (defaults to 95%) are optional. Our analysis estimates the likelihood that e is true, bounds any error in that estimate, and determines whether that estimate is significantly different from p. For example, consider the following function, which adds Gaussian noise to users' true locations to protect their privacy.

```
def obfuscate_location(location):
  noise = random.gauss(0,1)
  d = distance(location, location + noise)
  passert d < 10, 0.9, 95%
  return location + noise
```

To ensure that obfuscation does not change a user's true location too much, the programmer asserts that the Euclidean distance between the true and obfuscated location should be within 10 miles at least 90% of the time with 95% confidence. While occasional outliers are acceptable, the programmer wants to ensure that the common case is sufficiently accurate and therefore useful.

A traditional assertion—**assert** d $<$ 10—does not express this intent. Since the Gaussian distribution has a non-zero chance of adding any amount of noise, some executions will make d greater than 10. Since these infrequent outlier cases are possible, traditional verification must conclude that the assertion does not hold.

PLDI '14, June 9–11 2014, Edinburgh, United Kingdom.

http://dx.doi.org/10.1145/2594291.2594294

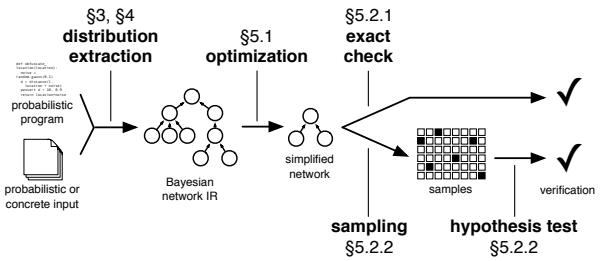


Figure 1. MAYHAP's workflow to verify probabilistic assertions.

Probabilistic Evaluation Probabilistic evaluation verifies the probabilistic logical statement over random variables expressed by the passert. It first performs *distribution extraction*, which is a symbolic execution that builds a Bayesian network, a directed, acyclic graphical model. Nodes represent random variables from the program and edges between nodes represent conditional dependences between those random variables. This process defines a probabilistic semantics in which *all* variables are distributions. Constants (e.g., $\mathsf{x} = 3$) are point-mass distributions. Random distributions, both simple (uniform, Gaussian, etc.) and programmer-defined, are represented symbolically. Other variables are defined in terms of these basic distributions.

For example, let L, D, and N be the random variables corresponding to the variables location, d, and noise in the above program. The passert constrains the probability $\Pr[D < 10]$ given that L is a point-mass distribution and that N is drawn from a Gaussian:

$$\Pr[D < 10 \mid L = \mathsf{location}, N \sim \mathcal{N}(0, 1)] > 0.9$$

This inequality constrains the probability of correctness for a particular input location. Alternatively, programmers may express a distribution over expected input locations by, for example, setting the location variable to sample from a uniform distribution. The passert then measures the likelihood that the obfuscation will yield acceptable results for uniformly distributed input locations:

$$\Pr[D < 10 \mid L \sim \mathcal{U}, N \sim \mathcal{N}(0, 1)] > 0.9$$

Our key insight is that, with this probabilistic semantics for passerts, we can optimize the Bayesian network representation and significantly improve the efficiency of verification. Using known statistical properties, our optimizations produce a simplified but equivalent Bayesian network. For example, we exploit identities of common probability distributions and Chebyshev's inequality. In some cases, these simplifications are sufficient to facilitate direct computation and verify the passert precisely. Otherwise, we sample the simplified Bayesian network and perform a hypothesis test to statistically verify the passert. We use *acceptance sampling*, a form of hypothesis testing, to bound the chance of both false positives and false negatives subject to a confidence level. Programmers can adjust the confidence level to trade off between analysis time and verification accuracy.

Evaluation We implement this approach in a tool called MAYHAP that takes C and C++ programs with passerts as inputs. MAYHAP emits either *true*, *false*, or *unverifiable* along with a confidence interval on the assertion's probability. Figure 1 gives an overview. We implement the entire toolchain for MAYHAP in the LLVM compiler infrastructure [18]. First, MAYHAP transforms a probabilistic C/C++ program into a Bayesian network that expresses the program's probabilistic semantics. For program inputs, developers provide concrete values or describe input distributions. MAYHAP optimizes the Bayesian-network representation using statistical properties and then either evaluates the network directly or performs sampling.

We implement case studies from three application domains: sensors, data obfuscation, and approximate computing. We show that passerts express their correctness properties and that MAYHAP offers an average speedup of 24× over stress testing with rote sampling. MAYHAP's benefits over simple stress testing—repeated execution of the original program—are threefold. First, statistical simplifications to the Bayesian network representation reduce the work required to compute each sample: for example, reducing the sum of two Gaussian distributions into a single Gaussian halves the necessary number of samples. Second, distribution extraction has the effect of partially evaluating the probabilistic program to slice away the non-probabilistic parts of the computation. Sampling the resulting Bayesian network eliminates wasteful re-execution of deterministic code. Third, our approach either directly evaluates the passert or derives a number of samples sufficient for statistical significance. It thereby provides statistical guarantees on the results of sampling that blind stress testing does not guarantee.

Although programs that compute with probabilistic data are already ubiquitous, abstractions and tools to help their developers are lagging. By harnessing randomness, our approach introduces new and effective abstractions for correctness, optimization, and verification of probabilistic programs.

2. Programming Model

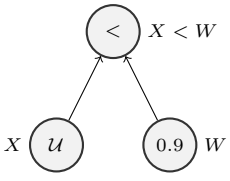
This section presents an intuitive view of programs as probabilistic computations over random variables. For our purposes, a probabilistic program is an ordinary imperative program that calls sampling functions for probability distributions [16]. Consider this simple program:

```
x = random.uniform(0,1)
w = 0.9
passert x < w, 0.90
```

This program samples from a uniform distribution, ranging from 0 to 1, assigns a concrete value to w, and then asserts that the sample is less than 0.9 using the comparison $\mathsf{x} < \mathsf{w}$ with 90% probability. An invocation of random.uniform returns one sample from the distribution. The language provides a library of sampling functions for common distributions, such as uniform, Gaussian, and Bernoulli distributions. Programmers may define sampling functions for new distributions using arbitrary code.

Programmers write specifications of correctness in passerts. The above passert is satisfied because the probability that a random sample from $\mathcal{U}(0, 1)$ is less than 0.9 is exactly 90%.

To formalize this reasoning, we represent programs as Bayesian networks. A Bayesian network is a directed, acyclic graphical model wherein nodes represent random variables and edges represent conditional dependence between those random variables.



Much like an expression tree, each node in the Bayesian network corresponds to a value produced by the program. Unlike an expression tree, however, each node represents a distribution rather than a single value. This network, for example, contains three random variables (X, W, and $X < W$), one for each expression executed in the program (random.uniform(0,1), 0.9, and $\mathsf{x} < \mathsf{w}$). The directed edges represent how these random variables conditionally depend on one another. For example, the node for the random variable $X < W$ has edges from two other nodes: X and W.

Because each variable is dependent *only* on its parents in a Bayesian network, the probability distributions for each node are defined locally. In our example, the distribution for the $X < W$ node, a Bernoulli random variable, is:

$$\Pr[X < W \mid X \sim \mathcal{U}, W = 0.9]$$

Computing the distribution for $X < W$ requires only the distributions for its parents, X and W. In this case, both parents are leaves in the Bayesian network: a uniform distribution and a point-mass distribution.

One way to compute the distribution is to sample it. Sampling the root node consists of generating a sample at each leaf and then propagating the values through the graph. Since Bayesian networks are acyclic, every node generates only a single value per sample and the running time of each sample is bounded.

In this example, we can exploit the Bayesian network formulation to simplify the graph and compute an exact solution without sampling. By definition, when X is uniformly distributed, for any constant $c \in [0, 1]$, $\Pr[X < c] = c$. Using this statistical knowledge, we replace the tree in our example with a single node representing a Bernoulli distribution with probability 0.9.

The Bayesian network abstraction for probabilistic programs yields two major advantages. First, it gives a probabilistic semantics to programs and passert statements. Section 4 formalizes our probabilistic semantics and proves that sampling the Bayesian representation is equivalent to sampling the original program. Second, we exploit probabilistic algebras and statistical properties of random variables to optimize the verification process. In some cases, we verify passerts without sampling. Section 5.1 introduces these optimizations.

3. Distribution Extraction

Given a program with a passert e and either a concrete input or a distribution over inputs, MAYHAP performs a probabilistic evaluation by building and optimizing a Bayesian-network representation of the statements required to evaluate the passert. This section describes distribution extraction, which is the first step in this process. Distribution extraction produces a symbolic Bayesian network representation that corresponds to the slice of the program contributing to e. MAYHAP treats randomness as symbolic and deterministic components as concrete. The process is similar to symbolic execution and to lazy evaluation in functional languages.

Distributions as Symbolic Values MAYHAP performs a forward pass over the program, concretely evaluating deterministic computations and introducing symbolic values—probability-distribution expression trees—to represent probabilistic values. For example, the following statement:

```
a = b + 2
```

computes a concretely when b is not probabilistic. If, prior to the above statement, the program assigns b = 5, then we perform the addition and set $a = 7$. However, if b = gaussian(), we add a node to the Bayesian network, representing b symbolically as a Gaussian distribution. We then create a sum node for a with two parents: b's Gaussian and 2's constant (point mass) distribution.

As this mixed symbolic and concrete execution proceeds, it eagerly evaluates any purely deterministic statements but builds a Bayesian-network representation of the forward slice of any probabilistic statements. This process embodies a symbolic execution in which the symbolic values are probability distributions. Our approach differs from typical symbolic execution in how it handles control flow (see below).

When the analysis reaches a statement passert e, MAYHAP records the Bayesian network rooted at e. It then optimizes the network and samples the resulting distribution. Compared to sampling the entire program repeatedly, sampling the extracted distribution can be more efficient even without optimizations since it eliminates redundant, non-probabilistic computation.

Conditionals When conditionals and loops are based on purely concrete values, distribution extraction proceeds down one side of the control flow branch as usual. When conditions operate on probabilistic variables, the analysis must capture the effect of both branch directions.

To analyze the probability distribution of a conditional statement, we produce conditional probabilities based on control-flow divergence. For example, consider this simple program:

```
if a: b = c else: b = d
```

in which a is probabilistic. Even if both c and d are discrete, the value of b is probabilistic since it depends on the value of a. We can write the conditional probability distributions $\Pr[B]$ for b conditioned on both possible values for a:

$$\Pr[B \mid A = \text{true}] = \Pr[C]$$
$$\Pr[B \mid A = \text{false}] = \Pr[D]$$

Instead, to simplify the representation of probabilities and to enable more straightforward analysis, we *marginalize* the condition a to produce an unconditional distribution for b. Using marginalization, we write the unconditional distribution $\Pr[B]$ as:

$$\begin{aligned}\Pr[B] &= \sum_a \Pr[B \mid A = a] \Pr[A = a] \\ &= \Pr[B \mid A = \text{true}] \Pr[A = \text{true}] \\ &\quad + \Pr[B \mid A = \text{false}] \Pr[A = \text{false}] \\ &= \Pr[C] \cdot \Pr[A = \text{true}] \\ &\quad + \Pr[D] \cdot (1 - \Pr[A = \text{true}])\end{aligned}$$

This expression computes the distribution for b as a function of the distributions for a, c, and d. Intuitively, the probabilistic evaluation rewrites the condition to read b = a * c + (1 − a) * d. This algebraic representation enables some optimizations described in Section 5.1.

Loops and External Code Loops with probabilistic conditions can, in general, run for an unbounded number of iterations. Representing unbounded execution would induce cycles in our graphical model and violate the acyclic definition of a Bayesian network. For example, consider a loop that accumulates samples and exits when the sum reaches a threshold:

```
v = 0.0
while v < 10.0:
  v += random.uniform(−0.5, 1.0)
```

If the random sample is negative in every iteration, then the loop will never exit. The probability of this divergence is small but non-zero.

Prior work has dealt with probabilistic loops by restricting the program to linear operators [33]. MAYHAP relaxes this assumption by treating a loop as a black box that generates samples (i.e., the loop may run for an unbounded but finite number of iterations), similar to a known probability distribution such as random.uniform. This representation avoids creating cycles. In particular, MAYHAP represents a loop body with a *summary node*, where variables read by the loop are edges *into* the node and variables written by the loop are edges *out* of the node.

In practice, many loops in probabilistic programs have non-probabilistic bounds. For example, we evaluated an image filter (sobel) that loops over the pixel array and applies a probabilistic convolution to each window. The nested loops resemble:

```
for x in 0..width:
  for y in 0..height:
    filter(image[x][y])
```

$$P \equiv S\ ;;\ \textbf{passert}\ C$$
$$C \equiv E < E \mid E = E \mid C \wedge C \mid C \vee C \mid \neg C$$
$$E \equiv E + E \mid E * E \mid E \div E \mid R \mid V$$
$$S \equiv V := E \mid V \leftarrow D \mid S\ ;\ S \mid \textbf{skip} \mid \textbf{if}\ C\ S\ S \mid \textbf{while}\ C\ S$$
$$R \in \mathbb{R},\ V \in \text{Variables},\ D \in \text{Distributions}$$

Figure 2. Syntax of PROBCORE.

While the computed pixel array contains probabilistic data, the dimensions width and height are fixed for a particular image. MAYHAP extracts complete distributions from these common concrete-bounded loops without black-box sampling.

MAYHAP uses a similar black-box mechanism when interfacing with library code whose implementation is not available for analysis—for example, when passing a probabilistic value to the cos() function from the C standard library. This straightforward approach prevents statistical optimizations inside the library function or loop body but lets MAYHAP analyze more programs.

Analyzing Loops with Probabilistic Path Pruning We propose another way to analyze loops with probabilistic bounds by building on the path pruning techniques used in traditional symbolic execution. Typically, path pruning works by proving that some paths are infeasible. If the analysis determines that a path constraint is unsatisfiable, it halts exploration of that path. Probabilistic evaluation instead needs to discover when a given path is *unlikely* rather than impossible, i.e., when the conditions that lead to following this path at run time have a probability that falls below a threshold. We propose tracking a path probability expression for each explored path and periodically sampling these distributions to prune unlikely paths. This extension handles general probabilistic control flow in programs that are likely to terminate eventually. Intuitively, the more iterations the loop executes, the less likely it is to execute another iteration. Programs with a significant probability of running forever before reaching a passert can still prevent the analysis from terminating, but this behavior likely indicates a bug. We leave the evaluation of this more precise analysis to future work.

4. Distribution Extraction Formalism

This section formalizes a simple probabilistic imperative language, PROBCORE, and MAYHAP's distribution extraction process. We describe PROBCORE's syntax, a *concrete semantics* for nondeterministic run-time execution, and a *symbolic semantics* for distribution extraction. Executing a PROBCORE program under the symbolic semantics produces a Bayesian network for a passert statement. We prove this extracted distribution is equivalent to the original program under the concrete semantics, demonstrating the soundness of MAYHAP's core analysis.

4.1 Core Language

PROBCORE is an imperative language with assignment, conditionals, and loops. Programs use probabilistic behavior by sampling from a distribution and storing the result, written $v \leftarrow D$. Without loss of generality, a program is a sequence of statements followed by a single passert, since we may verify a passert at any program point by examining the program prefix leading up to the passert.

Figure 2 defines PROBCORE's syntax for programs denoted P, which consist of conditionals C, expressions E, and statements S. For example, we write the location obfuscator from earlier as:

```
locationX ← Longitude; locationY ← Latitude;
noiseX ← Gauss[0, 1]; noiseY ← Gauss[0, 1];
newX := locationX + noiseX; newY = locationY + noiseY;
dSquared := ((locationX − newX) ∗ (locationY − newY))
  + ((locationY − newY) ∗ (locationY − newY));;
passert dSquared < 100
```

We draw the Longitude and Latitude inputs from opaque distributions and noise from Gauss[0, 1]. The entirety of Gauss[0, 1] is an opaque label; 0 and 1 are not expressions in our simple language.

4.2 Concrete Semantics

The concrete semantics for PROBCORE reflect a straightforward execution in which each sampling statement $V \leftarrow D$ draws a new value. To represent distributions and sampling, we define distributions as functions from a sufficiently large set of *draws* $\mathcal{S}$. The draws are similar to the seed of a pseudorandom number generator: a sequence Σ of draws dictates the probabilistic behavior of PROBCORE programs.

We define a large-step judgment $(H, e) \Downarrow_c v$ for expressions and conditions and a small-step semantics $(\Sigma, H, s) \rightarrow_c (\Sigma', H', s')$ for statements. In the small-step semantics, the heap H consists of the variable-value bindings (queried with $H(v)$) and Σ is the sequence of draws (destructed with $\sigma : \Sigma'$). The result of executing a program is a Boolean declaring whether or not the condition in the passert was satisfied at the end of this particular execution.

The rules for most expressions and statements are standard. The rules for addition and assignment are representative:

$$\textsc{plus}\ \frac{(H, e_1) \Downarrow_c v_1 \qquad (H, e_2) \Downarrow_c v_2}{(H, e_1 + e_2) \Downarrow_c v_1 + v_2}$$

$$\textsc{assign}\ \frac{(H, e) \Downarrow_c x}{(\Sigma, H, v := e) \rightarrow_c (\Sigma, (v \mapsto x) : H, \textbf{skip})}$$

The full semantics appear in this paper's auxiliary material [31], but we highlight the rule for sample statements here. The rule for the sampling statement, $V \leftarrow D$, consumes a draw σ from the head of the sequence Σ. It uses the draw to compute the sample, $d(\sigma)$.

$$\textsc{sample}\ \frac{\Sigma = \sigma : \Sigma'}{(\Sigma, H, v \leftarrow d) \rightarrow_c (\Sigma', (v \mapsto d(\sigma)) : H, \textbf{skip})}$$

The result of an execution under the concrete semantics is the result of the passert condition after evaluating the program body. We use the standard definition of $\rightarrow_c^*$ as the reflexive, transitive closure of the small step judgment:

$$\textsc{passert}\ \frac{(\Sigma, H_0, s) \rightarrow_c^* (\Sigma', H', \textbf{skip}) \qquad (H', c) \Downarrow_c b}{(\Sigma, H_0, s\ ;;\ \textsf{passert}\ c) \Downarrow_c b}$$

4.3 Symbolic Semantics

While the concrete semantics above describe PROBCORE program execution, the symbolic semantics in this section describe MAYHAP's distribution extraction. Values in the symbolic semantics are expression trees that represent Bayesian networks. The result of a symbolic execution is the expression tree corresponding to the passert condition, as opposed to a Boolean.

The language for expression trees includes conditions denoted C_o, real-valued expressions E_o, constants, and distributions:

$$C_o \equiv E_o < E_o \mid E_o = E_o \mid C_o \wedge C_o \mid C_o \vee C_o \mid \neg C_o$$
$$E_o \equiv E_o + E_o \mid E_o * E_o \mid E_o \div E_o \mid R \mid \langle D, E_o \rangle \mid \textbf{if}\ C_o\ E_o\ E_o$$
$$R \in \mathbb{R},\ D \in \text{Distributions}$$

Instead of the stream of draws Σ used in the concrete semantics, the symbolic semantics tracks a stream offset and the distribution D for every sample. Different branches of an **if** statement can

sample a different number of times, so the stream offset may depend on a conditional; thus, the stream offset in $\langle d, n\rangle$ is an expression in E_o and not a simple natural number. The symbolic semantics does not evaluate distributions, so the draws themselves are not required. Expression trees do not contain variables because distribution extraction eliminates them.

The symbolic semantics again has big-step rules $\Downarrow_s$ for expressions and conditions and small-step rules $\rightarrow_s$ for statements. Instead of real numbers, however, expressions evaluate to expression trees in E_o and the heap H maps variables to expression trees. For example, the rules for addition and assignment are:

$$\textsc{plus}\ \frac{(H, e_1) \Downarrow_s \{x_1\} \qquad (H, e_2) \Downarrow_s \{x_2\}}{(H, e_1 + e_2) \Downarrow_s \{x_1 + x_2\}}$$

$$\textsc{assign}\ \frac{(H, e) \Downarrow_s \{x\}}{(n, H, v := e) \rightarrow_s (n, (v \mapsto \{x\}) : H, \mathbf{skip})}$$

The syntax $\{x\}$ represents an expression in E_o, with the brackets intended to suggest quotation or suspended evaluation.

The rule for samples produces an expression tree that captures the distribution and the current stream offset:

$$\textsc{sample}\ \frac{}{(n, H, v \leftarrow d) \rightarrow_s (n + 1, (v \mapsto \{\langle d, n\rangle\}) : H, \mathbf{skip})}$$

Each sample statement increments the stream offset, uniquely identifying a sample expression tree. This enumeration is crucial. For example, enumerating samples distinguishes the statement $x \leftarrow d;\ y := x + x$ from a similar program using two samples: $x_1 \leftarrow d;\ x_2 \leftarrow d;\ y := x_1 + x_2$. This approach to numbering samples resembles *naming* in Wingate et al. [36].

The symbolic semantics must consider both sides of an **if** statement. For each **if** statement, we need to merge updates from both branches and form conditional expression trees for conflicting updates. We introduce a function `merge`, which takes two heaps resulting from two branches of an **if** along with the condition and produces a new combined heap. Each variable that does not match across the two input heaps becomes an $\{\mathbf{if}\ c\ a\ b\}$ expression tree in the output heap. The definition of `merge` is straightforward and its post-conditions are:

$$\frac{H_t(v) = a \qquad H_f(v) = b \qquad a \neq b}{\texttt{merge}(H_t, H_f, \{x\})(v) = \{\mathbf{if}\ x\ a\ b\}}$$

$$\frac{H_t(v) = a \qquad H_f(v) = b \qquad a = b}{\texttt{merge}(H_t, H_f, \{x\})(v) = a}$$

Using the `merge` function, we write the rule for **if** statements:

$$\textsc{if}\ \frac{(H, c) \Downarrow_s \{x\} \qquad (H, b_t) \rightarrow_s^* (H_t, \mathbf{skip}) \qquad (H, b_f) \rightarrow_s^* (H_f, \mathbf{skip})}{(n, H, \mathbf{if}\ c\ b_t\ b_f) \rightarrow_s (n, \texttt{merge}(H_t, H_f, \{x\}), \mathbf{skip})}$$

Our symbolic semantics assumes terminating **while** loops. Symbolic execution of potentially-unbounded loops is a well-known problem and, accordingly, our formalism only handles loops with non-probabilistic conditions. A simple but insufficient rule for **while** is:

$$\textsc{while}\ \frac{}{(n, H, \mathbf{while}\ c\ s) \rightarrow (n, H, \mathbf{if}\ c\ (\mathbf{while}\ c\ s))}$$

This rule generates infinite expression trees and prevents the analysis from terminating. We would like our analysis to exit a loop if it can prove that the loop condition is false—specifically, when the condition does not depend on any probability distributions. To capture this property, we add the following rule:

$$\textsc{while0}\ \frac{(H, c) \Downarrow_s \{x\} \qquad \forall \Sigma, (\Sigma, \{x\}) \Downarrow_o \mathbf{false}}{(n, H, \mathbf{while}\ c\ s) \rightarrow (n, H, \mathbf{skip})}$$

Here, the judgment $(\Sigma, \{x\}) \Downarrow_o v$ denotes evaluation of the expression tree $\{x\}$ under the draw sequence Σ. This rule applies when MAYHAP proves that an expression tree evaluates to **false** independent of the random draws. In our implementation, MAYHAP proves simple cases, when an expression tree contains no samples, and uses black-box sampling otherwise. Section 3 describes a more precise analysis that bounds path probabilities, but we leave its formalization to future work.

We can now define the symbolic evaluation of programs:

$$\textsc{passert}\ \frac{(0, H_0, s) \rightarrow_s^* (n, H', \mathbf{skip}) \qquad (H', c) \Downarrow_s \{x\}}{(H_0, s\ ;;\ \mathsf{passert}\ c) \Downarrow_s \{x\}}$$

To evaluate the resulting expression tree requires a sequence of draws Σ but no heap. The full set of rules are in the auxiliary material [31]; the rules for addition and sampling are representative:

$$\textsc{plus}\ \frac{(\Sigma, e_1) \Downarrow_o v_1 \qquad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 + e_2) \Downarrow_o v_1 + v_2} \qquad \textsc{sample}\ \frac{}{(\Sigma, \langle d, k\rangle) \Downarrow_o d(\sigma_k)}$$

The purpose of this formalism is to show that using MAYHAP's distribution extraction and sampling the result is equivalent to sampling the original program. We state this as a theorem for finite expression trees.

Theorem 1. *Let* $(0, H, p) \Downarrow_s \{x\}$*, where* x *is a finite program. Then* $(\Sigma, H_0, p) \Downarrow_c b$ *if and only if* $(\Sigma, x) \Downarrow_o b$*.*

In other words, for any draw sequence Σ, a program and its extracted distribution evaluate to the same output. Sampling the two distributions—executing them many times with different values for Σ—yields the same probability. Intuitively, the theorem is true because the symbolic semantics correspond to lazy evaluation and the output semantics $\Downarrow_e$ correspond to forcing of the resulting symbolic expressions. Formally, the proof of this theorem proceeds by a structural induction. Full details are in this paper's auxiliary material [31].

Future work can prove optimizations correct using the expression tree representation. Since we have proven equivalence with the original program representation, such correctness proofs can avoid dealing with variables and control flow, which are eliminated during distribution extraction.

5. The Rest of Probabilistic Evaluation: Optimization and Evaluation

To verify a conditional in a passert, probabilistic evaluation extracts a symbolic representation of the conditional, optimizes this representation, and evaluates the conditional. The previous sections described the distribution extraction step and this section describes our optimization and evaluation steps.

Optimizations simplify the Bayesian network by applying known statistical properties to make verification more efficient. In restricted cases, these optimizations simplify the Bayesian network to a closed-form Bernoulli representing the condition in the passert and we thus evaluate the passert exactly. In the general case, we use sampling and hypothesis testing to verify it statistically.

5.1 Optimizing Bayesian Networks

This section enumerates the statistical properties that MAYHAP applies to simplify distributions.

Closed-Form Operations on Known Distributions MAYHAP exploits closed-form algebraic operations on the common Gaussian, uniform, and Bernoulli distributions. For example, if $X \sim N(\mu_x, \sigma_x^2)$ and $Y \sim N(\mu_y, \sigma_y^2)$ then $X + Y \sim N(\mu_x + \mu_y, \sigma_x^2 + \sigma_y^2)$. Likewise, if $X \sim N(\mu_x, \sigma_x^2)$ then $X + 3 \sim N(\mu_x + 3, \sigma_x^2)$. MAYHAP optimizes closed form addition of Gaussians and scalar shifts or scaling of Gaussians, uniforms, and Bernoullis. We note there are many distributions and operations which we do not yet encode (e.g., a sum of uniform distributions is an Irwin–Hall distribution). Expanding the framework to capture a larger catalog of statistical properties is left to future work.

Inequalities Over Known Distributions MAYHAP uses the cumulative distribution function (CDF) for known distributions to simplify inequalities. The CDF for a real-valued random variable X is the function F_X such that $F_X(x) = \Pr[X < x]$, which provides a closed-form mechanism to evaluate whether a distribution is less than a constant. For example, if $X \sim U(0, 1)$ and the programmer writes the inequality $X < 0.9$, we reduce the inequality to a Bernoulli because $F_{Uniform}(0.9) = \Pr[X < 0.9] = 0.9$.

Central Limit Theorem The sum of a large number of independent random variables with finite variance tends to a Gaussian. MAYHAP uses the Central Limit Theorem to reduce loops which compute a reduction over random variables into a closed-form Gaussian which samples from the body of the loop. This transformation resembles the mean pattern exploited by Misailovic et al. [23]. It is particularly effective on the sobel application used in our evaluation, which averages the errors for each pixel in an array. MAYHAP reduces this accumulation to a single Gaussian.

Expectation Propagation The prior optimizations all approximately preserve a program's semantics: the transformed Bayesian network is approximately equivalent to the original Bayesian network. However, using statistical laws that apply to inequalities over random variables, it suffices to instead compute only the expected value and variance of a distribution. MAYHAP uses this insight to further simplify Bayesian networks by exploiting (1) the linearity of expected value and (2) statistical properties of inequality.

First, MAYHAP uses the linearity of expectation to produce simpler distributions with the same expected value as the original distribution. This is an important optimization because verifying a passert amounts to calculating the expected value of its underlying Bernoulli distribution. For example, the Bayesian network for $D + D$, which computes two independent samples from D, is not equivalent to the Bayesian network induced from $2 \cdot D$. So an optimization resembling traditional strength reduction does not compute the correct distribution. However, these two Bayesian networks have the same expected value. Specifically, expectation has the property $\mathrm{E}[A + B] = \mathrm{E}[A] + \mathrm{E}[B]$ for all distributions A and B. When only the expected value is needed, MAYHAP optimizes $D + D$ to $2 \cdot D$. A similar property holds for variance when the random variables are uncorrelated.

The reasoning extends to comparisons via Chebyshev's inequality. Given the expectation μ and variance σ^2 of a random variable, Chebyshev's inequality gives a bound on the probability that a sample of a random variable deviates by a given number of standard deviations from its expected value. For example, for a program with **passert** x >= 5, distribution extraction produces a Bayesian network of the form $X \geq 5$. Using the linearity of expectation, say we statically compute that $\sigma = 3$ and $\mu = 1$ for X. Chebyshev's inequality states:

$$\Pr[|X - \mu| \geq k\sigma] \leq \frac{1}{k^2}$$

We want to bound the probability that $x \geq 5$. Since we have μ and σ, we can rewrite this condition as:

$$x \geq \mu + 2\sigma$$
$$x - \mu \geq 2\sigma$$

So the passert condition states that x deviates from its mean by at least 2 standard deviations. Using $k = 2$ in Chebyshev's inequality gives the bound:

$$\Pr[X \geq 5] \leq \frac{1}{2^2}$$

We now have a bound on the probability (and hence the expectation) of the inequality x >= 5.

5.2 Verification

This section describes how we use a simplified Bayesian network to verify passerts using (1) exact (direct) evaluation or (2) sampling and statistical hypothesis testing.

5.2.1 Direct Evaluation

In some cases, simplifications on the probability distribution are sufficient to fully evaluate a passert. For example, MAYHAP simplifies the sobel application in our evaluation to produce a distribution of the form $\sum_n D < c$. The Central Limit Theorem optimization replaces the sum with a Gaussian distribution, which then enables the inequality computation to produce a simple Bernoulli distribution with a known probability. When dealing with a single Bernoulli, no sampling is necessary. MAYHAP reports the probability from the simplified distribution.

5.2.2 Statistical Verification via Sampling

In the general case, optimizations do not completely collapse a probability distribution. Instead, MAYHAP samples the resulting distribution to estimate its probability.

MAYHAP uses acceptance sampling to bound any error in its verification [37]. All passert statements are logical properties over random variables and therefore Bernoulli random variables. Assume $X_i \sim \mathrm{Bernoulli}(p)$ is an independent sample of a passert where p is the *true* probability of the passert, the value MAYHAP is estimating. Let $X = X_1 + X_2 + \cdots + X_n$ be the sum of n independent samples of the passert and let the empirical expected value, $\mathrm{E}[X] = \overline{X} = X/n$, be an estimate of p.[1] To bound error in its estimate, MAYHAP computes $\Pr[\overline{X} \in [p - \epsilon, p + \epsilon]] \geq 1 - \alpha$. In words, it tests whether there is at most an α chance that MAYHAP's estimate of p is wrong. Otherwise, MAYHAP's estimate of p is within ϵ of the truth. A programmer can control the likelihood of a good estimate—or the *confidence*—by decreasing α. Likewise, a programmer can control the *accuracy* of the estimate by decreasing ϵ. Because MAYHAP uses sampling, it provides statistical guarantees by testing whether its confidence interval for $\overline{X}$ includes $p \pm \epsilon$. In concert, these parameters let a programmer trade off false-positives and false-negatives with sample size.

In particular, given α and ϵ, MAYHAP uses the two-sided Chernoff bound to compute n, the minimum number of samples required to satisfy a given level of confidence and accuracy [9]. The two-sided Chernoff bound is an upper-bound on the probability that an estimate, $\overline{X}$, deviates from its true mean, p:

$$\Pr\left[|\overline{X} - p| \geq \epsilon p\right] \leq 2e^{-\frac{\epsilon^2}{2+\epsilon} np}$$

The left-hand side of the equality is α by definition and the worst case (the most samples required) occurs when $p = 1$. Solving for n yields:

$$n \geq \frac{2 + \epsilon}{\epsilon^2} \ln \frac{2}{\alpha}$$

[1] This section uses $\overline{X}$ instead of $\mathrm{E}[X]$ for notational convenience.

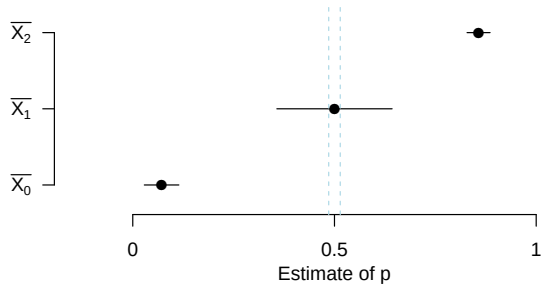


Figure 3. Hypothesis tests for three different passert statements.

For example, at a confidence 95% and an accuracy of 3%:

$$n \geq \frac{2 + 0.03}{0.03^2} \ln \frac{2}{0.05}$$

meaning that MAYHAP needs to take at least $n = 8321$ samples. Note that this bound is an over-approximation of the true number of samples required for a given level of confidence and accuracy—it only relies on α and ϵ and ignores how good an estimate $\overline{X}$ is of p. An extension, which we leave to future work, is to use Wald's *sequential sampling* to iteratively compute $\Pr\left[\overline{X} \in [p - \epsilon, p + \epsilon]\right] \geq 1 - \alpha$ after each sample [35]. Because this approach uses the current estimate of $\overline{X}$ relative to p, it is often able to stop sampling well before reaching our upper bound [38].

Statistical Guarantees The prior section describes how MAYHAP turns a passert statement into a hypothesis test in order to bound error in its estimate. If the property is sufficiently likely to hold, MAYHAP verifies the passert as true. Likewise, if the passert is verified as false, the programmer needs to iterate, either by changing the program to meet the desired specification or by correctly expressing the probabilistic property of the program.

For example, suppose MAYHAP estimates $\Pr[\overline{X}_i \in [p - \epsilon, p + \epsilon]] \geq 1 - \alpha$ for three distinct, hypothetical passert statements (i.e., $i \in [0, 1, 2]$). We pictorially show these three estimates in Figure 3. Each estimate shows $\overline{X}_i$ as a point and lines depict the confidence region of that estimate. Because the confidence region of $\overline{X}_0$ is below 0.5, MAYHAP verifies this assertion as false (i.e., the passert rarely holds). Likewise, because $\overline{X}_2 - \epsilon \geq 0.5$, MAYHAP verifies this assertion as true (i.e., the passert often holds).

However, at this confidence level and accuracy, MAYHAP is unable to verify $\overline{X}_1$ as its confidence region and thus estimate *overlaps* with $0.5 \pm \epsilon$. Thus, MAYHAP labels this assertion as unverifiable. To verify this assertion as true or false, the programmer must increase either the confidence or accuracy (or both). In this situation, MAYHAP initiates a dialog with the programmer for guidance on how to proceed.

6. Implementation

We implemented MAYHAP using the LLVM compiler infrastructure [18]. MAYHAP compiles source programs written in C and C++ to the LLVM intermediate language, probabilistically evaluates the resulting bitcode programs by extracting probability distributions, optimizes the resulting distributions, and then evaluates the passert distributions either exactly or with sampling.

Language and Interface To use MAYHAP, the programmer adds a passert to her program and annotates certain functions as probability distributions or uses a provided library of common distributions. Both constructs are implemented as C macros provided by a `passert.h` header: `PASSERT(e)` marks an expression that MAYHAP will evaluate and `DISTRIBUTION` marks functions that should be treated as a symbolic probability distribution.

The programmer invokes MAYHAP and provides the source files and command-line arguments for the execution along with optional α and ϵ values that control confidence and accuracy. MAYHAP reports a confidence interval on the output probability and the results of the hypothesis test (true, false, or unverifiable).

Distribution Extraction The distribution extraction analysis is implemented as an instrumented interpreter of LLVM bitcode programs. MAYHAP maintains a symbolic heap and stack. Each symbolic value is a pointer into an object graph representing a Bayesian network. Nodes in the graph correspond to the expression tree language of our formalism: they can be samples, arithmetic operations, comparisons, constants, or conditionals.

The implementation conserves space by coalescing identical expression trees. For example, consider the values $e_1 = \{s_1 + s_2\}$ and $e_2 = \{(s_1 + s_2) + s_3\}$ consisting of sums of samples. In a naive implementation of probabilistic evaluation, these would be independent trees that refer to a global set of samples at their leaves. Instead, MAYHAP implements e_2 as a sum node with two children, one of which is the node for e_1. In this sense, MAYHAP maintains a global Bayesian network for the execution in which values are pointers into the network.

Nodes in the Bayesian network can become unreachable when heap values are overwritten and as stack frames are popped. MAYHAP reclaims memory in these cases by reference-counting all nodes in the Bayesian network. The root set consists of stack and heap values. Since Bayesian networks are acyclic, reference counting is sufficient.

When operating on non-probabilistic values (e.g., when evaluating $1 + 2$), MAYHAP avoids constructing nodes in the Bayesian network and instead maintains a concrete heap and stack. We use the bitcode interpreter that ships with LLVM [20] to perform the concrete operations. This process can be viewed as an optimization on Bayesian networks for operations over point-mass distributions.

Conditionals Conditionals appear as branches in LLVM IR. MAYHAP analyzes conditionals by symbolically executing both sides of the branch and merging the resulting heap updates. When the analysis encounters a branch, it finds the immediate post-dominator (ipdom) in the control-flow graph—intuitively, the join point—and begins by taking the branch. In this phase, it buffers all heap writes in a hash table. Then, when the ipdom is reached, control returns to the branch and follows the not-taken direction. Writes in this phase are not buffered. When the ipdom is reached the second time, the buffered writes are merged into the heap using conditional nodes. MAYHAP supports nested conditions using a scoped hash table.

Probabilistic Pointers MAYHAP adds limited support for symbolic pointers for probabilistic array indexing. Programs can load and store from arr[i] where i is probabilistic, which MAYHAP handles with a probabilistic extension of the theory of arrays. Pointers and array indices must be finite discrete distributions so we can enumerate the set of locations to which a pointer p might refer, i.e., those addresses where p's distribution has non-zero probability. Loading from a symbolic pointer p yields a distribution that reflects the set of values at each such location, while storing to p updates each location to compose its old and new value under a conditional distribution.

Bayesian Network Optimizations MAYHAP performs statistical optimizations as transformations on the Bayesian network representation as outlined in Section 5.1. The optimizations we implemented fall into three broad categories, which we characterize empirically in the next section.

The first category consists of arithmetic identities, including binary operators on constants, comparisons with extremes (e.g.,

Program	Description and passert	Time (seconds) Baseline	Analysis	Sampling	Optimization Counts Arith	Dist Op	CLT
gpswalk	Location sensing and velocity calculation passert: Velocity is within normal walking speed	537.0	1.6	59.0	1914	0	1
salary	Calculate average of concrete obfuscated salaries passert: Obfuscated mean is close to true mean	150.0	2.5	< 0.1	3	1	1
salary-abs	salary with abstract salaries drawn from a distribution passert: As above	87.0	20.0	0.2	5003	1	1
kmeans	Approximate clustering passert: Total distance is within threshold	1.8	0.3	< 0.1	2149	300	0
sobel	Approximate image filter passert: Average pixel difference is small	37.0	2.8	< 0.1	7880	0	1
hotspot	Approximate CMOS thermal simulation passert: Temperature error is low	422.0	4.7	28.0	1	24064	1
inversek2j	Approximate robotics control passert: Computed angles are close to inputs	4.8	< 0.1	< 0.1	901	200	1

Table 1. Programs used in the evaluation. The passert for each application describes a probabilistic correctness property. The *time* columns indicate the time taken by the baseline stress-testing strategy, MAYHAP's analysis, and MAYHAP's sampling step. The *optimization counts* reflect the three categories of optimizations applied by MAYHAP: arithmetic identities (Arith), operations on known closed-form distributions (Dist Op), and the Central Limit Theorem optimization (CLT).

C's FLT_MAX), and addition or multiplication with a constant zero. These optimizations do not exploit the probabilistic properties of the Bayesian network but compose with more sophisticated optimizations and enhance the tool's partial-evaluation effect. The next category consists of operations on known probability distributions, including the addition of two normal distributions, addition or multiplication with a scalar, comparison between distributions with disjoint support, comparison between two uniform distributions, and comparison with a scalar (i.e., CDF queries). These optimizations exploit our probabilistic view of the program to apply well-known statistical properties of common distributions. The final optimization we evaluate is the Central Limit Theorem, which collapses a summation of distributions into a single normal.

Some optimizations, such as basic arithmetic identities, are performed opportunistically on-the-fly during analysis to reduce MAYHAP's memory footprint. Others, such as the Central Limit Theorem transformation, operate only on the complete graph. Internally, the on-line optimizer also collapses deep trees of commutative arithmetic operators into "fat" sum and product nodes with many children. This rewriting helps the optimizer identify constants that can be coalesced and inverse nodes that cancel each other out.

Verification via Direct Evaluation or Sampling As described in Section 5.2, the prior optimizations often produce Bayesian networks that MAYHAP can directly evaluate. In other cases, MAYHAP must sample the optimized Bayesian network, in which case MAYHAP generates LLVM bitcode that samples from the Bayesian network. The tool then compiles the generated program to machine code and executes it repeatedly to perform statistical verification.

7. Evaluation

This section describes our experience expressing passerts in a variety of probabilistic programs and using MAYHAP to verify them.

7.1 Benchmarks

We evaluate passerts in five probabilistic programs from three domains: sensors, differential privacy, and approximate computing. Table 1 summarizes the set of programs and the passert statements we added to each.

Programs that compute with noisy sensor data, such as GPS, accelerometers, and video game motion sensors, behave probabilistically [3, 26]. To demonstrate our approach on this class of applications, we implemented a common mobile-phone application: estimating walking speed [3]. gpswalk processes a series of noisy coordinate readings from a mobile phone and computes the walking speed after each reading. The GPS error follows a Rayleigh distribution and is determined by the sensor's uncertainty estimate. As Bornholt et al. [3] note, this kind of sensing workload can produce wild results when an individual location reading is wrong. The passert checks that the computed velocity is below a maximum walking speed.

Differential privacy obfuscates personal data at the cost of accuracy. To study how MAYHAP works on this class of application, we implemented two benchmarks. salary reads a list of 5000 salaries of Washington state public employees and computes their average.[2] The program obfuscates each salary by adding a normal distribution ($\sigma^2 = 3000$) to simulate a situation where each employee is unwilling to divulge their exact salary. The passert checks whether the obfuscated average is within 25 dollars of the true average. We also evaluate a version of the program, salary-abs, where the input salaries are drawn from a uniform distribution instead of read from a file. This variant highlights a scenario where specific inputs are unavailable and we instead want to check a passert given a input probability distribution.

The final class of applications is drawn from prior work on approximate computing: kmeans, sobel, hotspot, and inversek2j represent programs running on approximate hardware [7, 12, 25]. sobel implements the Sobel filter, an image convolution used in edge detection. kmeans is a clustering algorithm. hotspot simulates thermal activity on a microprocessor. inversek2j uses inverse kinematics to compute a robotic arm's joint angles given a target position. Both kmeans and hotspot are drawn from the Rodinia 2.4 benchmark suite [8] while sobel and inversek2j are approximate applications from Esmaeilzadeh et al. [11]. In all cases, we add random calls that simulate approximate arithmetic operations on inner computations. The passert bounds the error of the program's overall output. For most benchmarks, the error is measured with respect to the output

[2] Source: `http://fiscal.wa.gov/`

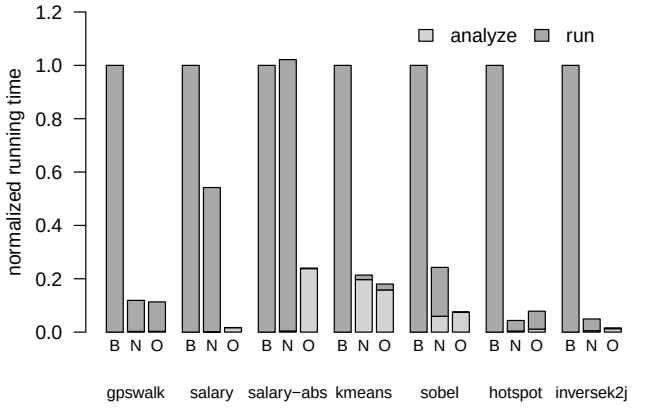


Figure 4. MAYHAP reduces testing time. We normalize to B: the baseline stress-testing technique with 74147 samples. N is MAYHAP without optimizations and O is MAYHAP with optimizations, divided into analysis and execution time. Times are averaged over 5 executions. We elide error bars as they are very small.

of a precise version of the computation, but in inversek2j, we use the corresponding forward kinematics algorithm to check the result.

For both approximate and data privacy programs, we compare a precise version of a function's output with a perturbed version. In the sensing workload, gpswalk, the data is intrinsically noisy, so there is no "ground truth" to compare against. For the purposes of this evaluation, we manually extended the programs to compute both results. A simple "desugaring" step could help perform this transformation automatically by duplicating the code, remove randomization from one copy, and return both results.

7.2 Performance

To evaluate MAYHAP's performance benefits, we compare its total running time against using a simple stress testing baseline. The baseline checker adds a **for** loop around the entire probabilistic program and counts the number of times the passert expression is true. The time taken for a MAYHAP verification includes the time to extract and optimize a probability distribution and to repeatedly sample the result. We test all programs with a confidence of $\alpha = 0.05$ and an accuracy of $\epsilon = 0.01$, which leads to 74147 samples. (Recall from Section 5.2.2 that the sample count depends only on the α and ϵ parameters and so we sample all programs the same number of times.) Table 1 lists the absolute running times and Figure 4 visualizes the normalized performance. The timings are averaged over 5 executions collected on a dual-core 2 GHz Intel Core 2 Duo with 4 GB of memory. On average, MAYHAP verification takes 4.2% as long as the strawman checker, for a speedup of 24×.

For most benchmarks, MAYHAP's time is almost exclusively spent on distribution extraction and optimization, which means optimizations are effective at producing a very small distribution that can be sampled much more efficiently than the original program. The exception is gpswalk, where the analysis executed in 1.6 seconds but sampling the resulting distribution took over a minute. That program's probability distribution consists of thousands of independent Rayleigh distributions, each with a different parameter as reported by the GPS sensor, so it cannot take advantage of optimizations that exploit many samples from identical distributions.

Effect of Optimizations We evaluated a variant of MAYHAP with optimizations disabled. This version simply performs distribution extraction and samples the resulting distribution. The middle bars labeled N in Figure 4 show the normalized running time of this non-optimizing MAYHAP variant.

The effectiveness of the optimizations varies among the benchmarks. On one extreme, optimizations reduce the execution time for salary from 81 seconds to a fraction of a second. The unoptimized Bayesian network for salary-abs is slightly *less* efficient than the original program. The Central Limit Theorem optimization applies to both and greatly reduces the amount of sampled computation. On the other hand, simply evaluating the extracted distribution delivers a benefit for gpswalk, reducing 537.0 to 62 seconds and then optimizations further reduce this time to just 59.0 seconds. In a more extreme case, enabling optimizations adds to the analysis time for hotspot but fails to reduce its sampling time. These programs benefit from eliminating the deterministic computations involved in timestamp parsing and distance calculation.

Confidence–Performance Trade-off Via the confidence and accuracy parameters α and ϵ, MAYHAP provides rough estimates quickly or more accurate evaluations using more samples. To evaluate this trade-off, we lowered the parameter settings, $\alpha = 0.10$ and $\epsilon = 0.05$, which leads to 2457 samples (about 3% compared to the more accurate settings above). Even accounting for analysis time, MAYHAP yields a harmonic mean 2.3× speedup over the baseline in this relaxed configuration.

8. Related Work

Researchers have proposed several languages and tools to help developers better reason about and describe real-world probabilistic data, computation, and models [2, 3, 6, 13, 14, 22, 27, 28, 36]. This section compares prior efforts with our work. At a high level, our approach lets programmers use traditional language features (e.g., calls to C's rand()) to express probabilistic semantics and a simple construct to encode probabilistic correctness properties.

Semantics and Verification of Probabilistic Programs The probability monad captures a variable's discrete probability distribution in functional programs [28]. Similarly, $Uncertain\langle T\rangle$ uses the monadic technique of building up a computation tree and then querying it and adds sampling and hypothesis testing to evaluate conditionals [3]. We build on this work by extending it to the problem of verification, by applying symbolic execution to summarize many program paths, and by adding the concept of optimization via statistical properties.

Sankaranarayanan et al. [33] check assertions in programs that produce probabilistic models using symbolic execution and polyhedral volume estimation. The estimateProbability construct queries the probability of an outcome, which resembles passert's specification of the correct outcome. That work's polyhedral approach avoids sampling but limits the technique in important ways: it works only with distributions for which a cumulative distribution function is available and programs that use only linear arithmetic over these distributions. Our distribution extraction approach uses sampling to generalize to a broader class of probabilistic programs.

Statistical model checking bounds verification error on problems where state-space explosion makes exact numerical verification intractable (see Legay and Delahaye's survey [19]). Model checking [10] provides formal guarantees, usually expressed in temporal logic for finite state based models, often of hardware. For example, the PRISM tool performs statistical verification of real-time systems [17]. Our work borrows the idea of hypothesis testing to bound error in verification [37, 38] and relies on efficient sampling to avoid the need for exhaustive state space exploration.

Kozen [16] recognizes the need for semantics for programs that use randomness during execution. That work provides two semantics for a simple probabilistic language—one that models sampling and one that computes on probability distributions directly—and proves them equivalent. Similarly, we prove equivalence between sampling an original program and sampling its extracted Bayesian network

representation. Kozen predates the coinage and popularization of Bayesian networks, so the semantics in that work are very different from the graphical-model approach presented here. Our Bayesian-network representation enables statistical optimizations that make passert verification efficient.

Probabilistic Programming The field of *probabilistic programming* seeks to enable efficient construction and querying of statistical models [2, 6, 13–15, 27, 36]. Experts write generative models as programs and then inference algorithms answer questions about the model's parameters. The canonical probabilistic programming example answers, "given that the grass is wet, was it due to rain or the sprinkler?"

In contrast, we focus on traditional computation of outputs based on user-specified inputs and do not incorporate *conditioning* (as in Infer.NET's constraints [22] and in Church's query evidence [13]). Our analysis instead supports general, potentially unanalyzable code that produces arbitrary probability distributions. These differences mean that our techniques apply to "probabilistic programming languages" in the more traditional sense as defined by Kozen [16]: typical imperative languages that include random calls.

Data Obfuscation for Privacy Recent work has focused on proving that obfuscated queries effectively obscure private data via *differential privacy* [1, 21, 24, 29, 30]. Probabilistic assertions, in contrast, do not check privacy. They instead solve the complementary (and less well-studied) problem of verifying the utility of private computations.

Approximate Computing Approximate computing techniques exploit the inherent resilience of many applications to execute them more efficiently at the cost of occasional errors [5, 11, 32, 34]. A central challenge in approximate computing is analyzing programs to determine the impact of approximation on a computation's overall accuracy. Previous efforts have used static analysis to prove statistical bounds on the difference between an original program and a version that elides some operations [23, 39]. Rather than analyzing probability distributions introduced by the program, this technique assumes that inputs are selected randomly and analyzes specific program patterns that involve them. In contrast, Rely [5] checks the probability that nondeterministic operations with a binary chance of failure compound to corrupt a computation's output. Carbin et al. [4] provide a system for proving properties that must hold even when errors occur. Our technique improves on these prior approaches by reasoning about the effects of full probability distributions on approximate programs.

Other Forms of Randomness Randomized algorithms solve computational problems probabilistically that are intractable to solve deterministically. Analyzing a randomized algorithm amounts to proving that its output satisfies a predicate with high probability, which resembles the guarantees given by checking probabilistic assertions, but is beyond our scope.

9. Conclusion

Programs use and compute with probabilistic data—be it big data or sensors or machine learning. Probabilistic programs are ubiquitous, yet we lack tools and analyses to help programmers understand their meaning. This paper embraces randomness and demonstrates how to represent arbitrary programs as a Bayesian network and thus give them a well-defined, probabilistic semantics. Programmers then express properties of probabilistic variables in a passert. We introduce *probabilistic evaluation* which extracts distributions from programs, optimizes them with algebras over probability distributions, and then verifies them directly or with hypothesis testing. Case studies on three application domains show that probabilistic evaluation can verify important correctness properties and that our approach is orders of magnitude more efficient than stress testing.

By exposing approximate quality conditions, we create a formalism for principled but approximate transformations. Akin to the way that dataflow formalism created a rigorous and fertile foundation for traditional compiler optimization of deterministic programs, this or some other probabilistic formalism should prove fertile for compiler optimization of probabilistic programs.

Acknowledgments

We would like to thank our anonymous reviewers for their invaluable comments. Our thanks also to Tom Bergan, Colin Gordon, and James Bornholt for feedback on earlier versions of this paper. This work was supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. It was also supported by the Qualcomm Innovation Fellowship and the Google PhD Fellowship.

References

[1] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *POPL*, 2012.

[2] S. Bhat, J. Borgström, A. D. Gordon, and C. Russo. Deriving probability density functions from probabilistic functional programs. In *TACAS*. Springer, 2013.

[3] J. Bornholt, T. Mytkowicz, and K. S. McKinley. Uncertain<T>: A first-order type for uncertain data. In *ASPLOS*, 2014.

[4] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, 2012.

[5] M. Carbin, S. Misailovic, and M. Rinard. Verifying quantitative reliability of programs that execute on unreliable hardware. In *OOPSLA*, 2013.

[6] A. T. Chaganty, A. V. Nori, and S. K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *AISTATS*, 2013.

[7] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee. Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology. In *DATE*, 2006.

[8] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*, 2009.

[9] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *The Annals of Mathematical Statistics*, 23(4):493–507, 1952.

[10] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Workshop on Logic of Programs*, pages 52–71, 1982.

[11] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[12] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, 2012.

[13] N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A language for generative models. In *UAI*, 2008.

[14] O. Kiselyov and C.-C. Shan. Embedded probabilistic programming. In *Working Conference on Domain-Specific Languages*, 2009.

[15] D. Koller, D. McAllester, and A. Pfeffer. Effective Bayesian inference for stochastic programs. In *AAAI*, 1997.

[16] D. Kozen. Semantics of probabilistic programs. In *Symposium on Foundations of Computer Science*, pages 101–114, Oct 1979.

[17] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. *CAV*, pages 585–591, 2011.

[18] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, 2004.

[19] A. Legay and B. Delahaye. Statistical model checking: A brief overview. *Quantitative Models: Expressiveness and Analysis*, 2010.

[20] LLVM Project. LLVM interpreter, 2013. `http://llvm.org/docs/doxygen/html/classllvm_1_1Interpreter.html`.

[21] F. D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *SIGMOD*, 2009.

[22] T. Minka, J. Winn, J. Guiver, and D. Knowles. Infer.NET 2.5, 2012. Microsoft Research Cambridge. `http://research.microsoft.com/infernet`.

[23] S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *SAS*, 2011.

[24] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler. GUPT: Privacy preserving data analysis made easy. In *SIGMOD*, 2012.

[25] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones. Scalable stochastic processors. In *DATE*, 2010.

[26] S. Park, F. Pfenning, and S. Thrun. A probabilistic language based upon sampling functions. In *POPL*, 2005.

[27] A. Pfeffer. A general importance sampling algorithm for probabilistic programs. Technical Report TR-12-07, Harvard University, 2007. `ftp://ftp.deas.harvard.edu/techreports/tr-12-07.pdf`.

[28] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *POPL*, 2002.

[29] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ICFP*, 2010.

[30] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *NSDI*, 2010.

[31] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Probabilistic assertions: Extended semantics and proof. ACM Digital Library auxiliary materials accompanying this paper. `http://dx.doi.org/10.1145/2594291.2594294`.

[32] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[33] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *PLDI*, 2013.

[34] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*, 2011.

[35] A. Wald. Sequential tests of statistical hypotheses. *The Annals of Mathematical Statistics*, 16(2):117–186, 1945.

[36] D. Wingate, A. Stuhlmüller, and N. D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Artificial Intelligence and Statistics*, 2011.

[37] H. Younes. Error control for probabilistic model checking. *Verification, Model Checking, and Abstract Interpretation*, pages 142–156, 2006.

[38] H. L. Younes and R. G. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. *Information and Computation*, 204(9):1368 – 1409, 2006.

[39] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, 2012.