We propose trace abstraction modulo probability, a proof technique for verifying high-probability accuracy guarantees of probabilistic programs. Our proofs overapproximate the set of program traces using failure automata, finite-state automata that upper bound the probability of failing to satisfy a target specification. We automate proof construction by reducing probabilistic reasoning to logical reasoning: we use program synthesis methods to select axioms for sampling instructions, and then apply Craig interpolation to prove that traces fail the target specification with only a small probability. Our method handles programs with unknown inputs, parameterized distributions, infinite state spaces, and parameterized specifications. We evaluate our technique on a range of randomized algorithms drawn from the differential privacy literature and beyond. To our knowledge, our approach is the first to automatically establish accuracy properties of these algorithms.

1 INTRODUCTION

With the recent explosion of interest in data analysis, randomized algorithms are increasingly seeing applications across all of computer science. These algorithms satisfy a wide variety of subtle probabilistic properties: modeling statistical privacy of database queries [Dwork and Roth 2014], stability and generalization of machine learning procedures [Bousquet and Elisseeff 2002], fairness of decision-making algorithms [Dwork et al. 2012], and more.

Many probabilistic properties are tailored to specific applications, but perhaps the most fundamental properties are accuracy guarantees, the probabilistic analogue of functional correctness. While such specifications would ideally hold all of the time—with probability 1—such stringent guarantees rule out many useful applications of randomization. Accuracy properties are often phrased as high-probability guarantees: for all program inputs, an output sampled from the final distribution satisfies \( \varphi \) except with some probability \( \beta \). For instance, a noisy numeric output \( r_{\text{noisy}} \) might satisfy a precision bound \( \varphi \triangleq |r_{\text{noisy}} - r_{\text{exact}}| < 5.2 \) except with probability at most \( \beta \triangleq 0.01 \), where \( r_{\text{exact}} \) is the answer without noise. While the failure probability \( \beta \) can be a concrete value in \([0, 1]\), it is often treated symbolically so that the guarantee \( \varphi \) may depend on \( \beta \)—e.g., \( \varphi(\beta) \triangleq |r_{\text{noisy}} - r_{\text{exact}}| < 1/\beta \) gives higher confidence guarantees by widening the error range. This kind of property describes how well an algorithm will perform at varying levels of confidence, crucial information for the algorithm designer. Our goal is to enable algorithm designers to prove accuracy specifications fully automatically.

While simple to state, accuracy guarantees—like other probabilistic properties—pose interesting challenges for automated verification. Current techniques have focused on more tractable models of randomized computation, especially probabilistic automata and Markov Decision Processes. (Recent surveys by Baier et al. [2018] and Katoen [2016] provide a good overview.) By treating parameters and inputs as known constants, tools can apply numerical methods to compute event probabilities in the output distribution. There are now several mature verification tools (e.g., [Dehnert et al. 2017; Kwiatkowska et al. 2011]), which have found notable success in helping designers automatically...
analyze complex probabilistic systems rigorously. However, their common foundation leads to common weaknesses: they are mostly restricted to closed programs with fixed inputs and finite state spaces, and support for properties with symbolic parameters remains limited.

In this paper, we start from established automated verification techniques for non-probabilistic programs and extend them to the probabilistic setting. Our logic-based approach yields several benefits. By reasoning symbolically instead of numerically, we can (i) directly establish properties for all inputs rather than requiring fixed inputs, (ii) handle programs that sample from distributions with unknown parameters, possibly over infinite ranges, and (iii) prove parametric accuracy properties, making it possible to automatically establish tradeoffs between accuracy and failure probabilities, and capture the dependence on other input parameters.

1.1 An Overview of Our Approach

Trace Abstraction Modulo Probability. Our approach is based on trace abstraction [Farzan et al. 2013; Heizmann et al. 2009, 2010, 2013], a proof technique for non-probabilistic verification. A program $P$ is represented by a language $L(P)$ of syntactic execution traces through the control-flow graph. To prove that $P$ satisfies $\varphi$, we first overapproximate $L(P)$ with finite-state automata:

$$L(P) \subseteq L(A_1) \cup \cdots \cup L(A_n)$$

To show that all traces in $\bigcup_i L(A_i)$ satisfy $\varphi$, the automata are annotated with Hoare-style assertions overapproximating reachable states along execution traces—annotations are usually computed via predicate abstraction [Graf and Saïdi 1997] or Craig interpolation [Craig 1957]. Intuitively, trace abstraction constructs proofs for subsets of traces; the combined proofs verify the whole program.

To extend this idea to probabilistic programs, suppose we want to prove that $\varphi$ holds except with probability at most $\beta$. We construct a set of automata that (i) overapproximates all traces in $L(P)$ and (ii) satisfies the following probabilistic bound:

$$\sum_{\tau \in \bigcup_i L(A_i)} \Pr[\varphi \text{ not true after running } \tau] \leq \beta$$

In words, the total failure probability—across all traces represented by the automata—is at most $\beta$. While trace inclusion is relatively simple to check, verifying the probabilistic bound over infinitely many traces is more challenging. To ease the task, our automata are annotated with Hoare-style assertions describing reachable program states, and also failure probabilities, upper bounds on the probability of not reaching those program states.

Automating Proofs. To construct trace abstractions automatically, our approach proves that individual program traces $\tau$ satisfy $\varphi$ with a failure probability $\beta$, and then generalizes the proof into a (potentially infinite) set of traces represented by an automaton $A_{\tau}$ such that the sum of failure probabilities of traces in $A_{\tau}$ is at most $\beta$. We repeatedly pick, prove, and generalize program traces until we have overapproximated the set of all possible traces $L(P)$.

The most technically intricate piece of our algorithm is the proving step. Given a trace $\tau$, we want to automatically prove that it satisfies $\varphi$ with failure probability $\beta$. In the non-probabilistic setting, trace semantics can be encoded as a logical formula and verification conditions can be discharged with an SMT solver. In our setting, however, traces have probabilistic semantics and a naïve encoding would be prohibitively complex. Instead, we reduce probabilistic reasoning to logical reasoning. Specifically, we encode the verification condition as a constraint-based synthesis problem of the form $\exists f. \forall X. \varphi$, where the function $f$ chooses between different axiomatizations of probability distributions that are sampled along the given trace. This choice affects the failure probability of the entire computation, and also determines what we can assume about the result of
the random sampling statements later on in the trace. The axioms can be seen as approximating the semantics of probabilistic samplings using a first-order theory amenable for checking by smt.

After proving correctness of a trace \( \tau \) by solving the formula \( \exists f. \forall X. \varphi \) for \( f \), we demonstrate how to use Craig interpolation—a well-studied proof technique in traditional, non-probabilistic verification—to construct a Hoare-style proof of the trace along with failure probabilities. This proof can then be generalized to cover a potentially infinite set of program traces.

**Implementation & Case Studies.** We have implemented our algorithm and applied it to a range of sophisticated randomized algorithms, mainly from the differential privacy literature [Dwork et al. 2006]. In differential privacy, algorithm designers must add noise to protect personal information, but try to guarantee good accuracy for their data analyses. Our technique automatically proves intricate accuracy specifications capturing this tradeoff. To demonstrate our approach’s generality, we apply our technique to reason about reliability of programs running on approximate hardware with probabilistic failures.

### 1.2 Design Rationale: Expressivity vs. Automation

We chose to solve our probabilistic verification problem by extending the trace abstraction framework, which has proved successful for traditional automated verification [Heizmann et al. 2018], to the probabilistic setting. We made this design choice for three interrelated reasons: First, the trace-based view of the verification problem lends itself cleanly to our probabilistic setting, as failure probability combines additively across (and along) traces. Second, as we demonstrate, reasoning about one finite trace at a time allows us to reduce probabilistic reasoning to logical reasoning. Third, by eliminating probabilistic reasoning, we can establish probabilistic properties by synthesizing rich annotations of failure automata in first-order logic, analogously to interpolation-based verification.

Avoiding probabilistic assertions means that our technique loses some expressivity. For example, we cannot take advantage of more precise analyses, such as concentration bounds based on independence of random variables. However, reducing probabilistic reasoning allows us to exploit established automated reasoning technologies—e.g., smt solvers and interpolation—which have been profitable in the non-probabilistic setting. Additionally, despite the loss in expressivity, we show that our technique captures most proofs in ahl [Barthe et al. 2016b], as discussed in § 5.2.

It is important to note that there are richer techniques for reasoning about probabilistic programs, like the seminal pre-expectation calculus [Kozen 1985; Morgan et al. 1996], which uses a probabilistic analogue of the weakest precondition transformer. Unfortunately, for complex distributions, like the infinite-support Laplace distribution, pre-expectations yield correspondingly complex integrals that are difficult to reason about automatically. Further, techniques that attempt to automate construction of expectation invariants (e.g., via Martingales [Chakarov and Sankaranarayanan 2013]) are limited in expressivity and require side probabilistic reasoning to establish a postcondition. We provide a more detailed comparison with probabilistic verification techniques in § 8.

### 1.3 Outline and Contributions

After demonstrating our verification strategy on two worked examples (§ 2) and introducing the program model (§ 3), we offer the following technical contributions.

- **Trace abstraction modulo probability (§ 4):** We present a proof rule for probabilistic accuracy properties, extending trace abstraction to the probabilistic setting. Our proof technique is based on the new notion of failure automata, labeled automata overapproximating program traces and their probability of failing to satisfy a given postcondition.
Automating trace-based proofs (§ 5): We present an algorithm for constructing trace-based proofs by iteratively building failure automata, proving correctness of finite program traces, and then generalizing the automata to cover potentially infinite sets of traces.

Proofs & interpolation for probabilistic traces (§ 6): We show that we can prove correctness of individual probabilistic program traces via a reduction to a constraint-based synthesis problem, making probabilistic reasoning unnecessary. Then, we demonstrate how to apply Craig interpolation to construct failure automata.

Implementation & case studies (§ 7): We implement our approach and use it to automatically prove accuracy guarantees of a range of randomized algorithms from the theory of differential privacy. We also study an example from the approximate computing literature. Our implementation establishes accuracy properties with symbolic parameters for programs with parametric inputs and infinite states, a first for automated verification.

Finally, we survey related work (§ 8) and conclude (§ 9).

2 OVERVIEW AND ILLUSTRATION

In this section, we provide an overview of our proof technique on two simple examples.

2.1 High-Level Overview

Suppose we are given a probabilistic program $P$, pre- and post-conditions $\phi_{\text{pre}}$ and $\phi_{\text{post}}$, and a numeric expression $\beta$ representing the maximum allowed failure probability. Our goal is to prove that if we start executing $P$ from any state satisfying $\phi_{\text{pre}}$, the probability that the output state does not satisfy $\phi_{\text{post}}$ upon termination is at most $\beta$. This property is denoted by the following formula, reminiscent of a Hoare triple:

$$\vdash_\beta \{ \phi_{\text{pre}} \} P \{ \phi_{\text{post}} \}$$

Proof Rule. We view $P$ as a control-flow automaton whose language $L(P)$ is the set of all traces from the program’s entry location to its exit location. Our proof rule overapproximates $L(P)$ by a larger set of traces, represented by a set of finite automata $\mathcal{A}$, while ensuring that the total failure probability across all traces in $L(\mathcal{A})$ is at most $\beta$.

Automation. To apply our proof rule automatically, we apply an algorithmic technique summarized in Fig. 1. The technique repeatedly tries to (i) pick a program trace $\tau \in L(P)$ outside the approximation $L(\mathcal{A})$, (ii) prove that $\vdash_\beta \{ \phi_{\text{pre}} \} \tau \{ \phi_{\text{post}} \}$, i.e., the probability that the trace falsifies the Hoare triple is at most $\beta$, and then (iii) generalize the trace $\tau$ into an automaton $A_\tau$ encoding a set of traces with total failure probability at most $\beta$. Our approach succeeds if it constructs a set of automata $\mathcal{A}$ modeling all program traces $L(P)$, with total failure probability at most $\beta$. 

![Fig. 1. Main loop of verification algorithm](image-url)
2.2 Illustrative Example: Loop-free Program

To warm up, we consider the loop-free program in Fig. 2. The function ex1 takes a single \([0,1]\)-valued input \(p\) and returns a Boolean value \(y\). Our goal is to prove the following accuracy property:

\[
\mathcal{F}_p \{true\} \text{ex1}(p) \{\neg y\}
\]

In words, the program fails to return \(y = false\) with probability at most \(p\). This property can be established informally: (i) the probability that the program takes the then branch and returns \(y = true\) is \(0.5p\); (ii) the probability that it takes the else branch and returns \(y = true\) is \(0.25p\). Therefore, the failure probability is \(0.5p + 0.25p \leq p\).

**Illustrating Proof Artifacts.** We begin by describing the proof artifacts constructed by our approach. The program ex1 is presented as a control-flow automaton over the alphabet of program statements, as shown in the left side of Fig. 3. Edge labels of the form \([c]\) are guards (also known as assume statements) encoding possible branches of the conditional statement. Accepted traces start from the initial node \(in\) and end in the final, accepting node \(ac\).

Our verification approach focuses on one trace at a time. There are two possible traces in our example program: one through the then branch and one through the else branch of the conditional. We refer to these traces as \(\tau_1\) and \(\tau_2\), respectively.

To prove accuracy properties about each trace, our technique annotates traces with auxiliary information. Let us consider the annotated trace \(\tau_1\) in Fig. 2. Each node along the trace is annotated with two labels: (i) the top/blue label is a logical formula representing a set of reachable program states at that point (these can be viewed as Hoare-style annotations); (ii) the bottom/red label is an expression representing the probability that the program does not end up in the blue states. Consider node \(in\) from \(\tau_1\): it is labeled by \(true\) and \(0\), indicating that the probability of failing to arrive in a program state satisfying \(true\) is \(0\) (as expected). However, consider node \(ac\): it is labeled with \(\neg y\) and \(p\), indicating that the probability of failing to arrive in a state where \(y = false\) is at most \(p\). The other program trace \(\tau_2\), which traverses the other branch, is similar; the annotation of \(\tau_2\) demonstrates that its failure probability is at most \(0.5p\).

At this point we have considered all of the ex1’s traces. If we naïvely sum up their probabilities of failure to bound the total failure probability, we get a failure probability of at most \(p + 0.5p = 1.5p\), which is too weak—we wanted to prove an upper bound of \(p\). However, we can give a more precise analysis since the two traces consider two mutually disjoint events: one path assumes \(x\) is true and the other assumes \(x\) is false.
while the other assumes \( x \) is false. In this case, we can soundly take the maximum of the two failure probabilities, \( 0.5p \) and \( p \), arriving at a total failure probability of \( p \) and concluding the proof.\(^1\)

Given a labeled trace, it is relatively straightforward to check if the annotations are valid. However, constructing the annotations may not be so easy. The main challenge is selecting labels for the results of sampling instructions—the invariants are not fully determined by the program, and in general the proper choice depends on the target property we are trying to establish. For instance, it is also possible to give an alternative annotation of \( \tau_1 \), denoted \( \tau'_1 \) in Fig. 2. Node \( i \) is labeled with \( x \) and 0.5, indicating that the probability of not arriving in a state where \( x = true \) is at most 0.5.

This annotated trace illustrates another general feature of our analysis: failure probabilities sum up along traces. Intuitively, this principle corresponds to a basic property of probabilities called the union bound: \( \Pr(A \cup B) \leq \Pr(A) + \Pr(B) \) for any two events \( A \) and \( B \). In particular, if \( A \) and \( B \) are interpreted as bad events—events violating labels at different nodes—the probability of any failure occurring along a trace is at most the sum of the failure probabilities of individual steps. In \( \tau'_1 \), the probability of \( y = true \) at node ac of \( \tau'_1 \) is \( p \), so the final failure probability computed for this trace is \( 0.5 + p \). While this annotation in \( \tau'_1 \) is sound, it is too weak to prove our desired property.

### Encoding Trace Semantics.

Our technique cleanly separates probabilistic assertions into two pieces: a non-probabilistic component describing the state of program variables (the blue annotations in Fig. 3), and a single number summarizing the probabilistic part of the assertion (the red annotations in Fig. 3). As a result, we can reduce probabilistic reasoning to logical reasoning, allowing us to harness the power of SMT solvers and synthesis techniques.

To illustrate, we show how to construct trace labels for \( \tau_1 \). Our method proceeds in two steps. First, like in traditional verification-condition generation, we encode the semantics of trace \( \tau_1 \) and the specification as a logical formula, which, if valid, implies that \( \tau \in \{true\} \tau_1 \{\neg y\} \). Specifically, we construct the following verification condition:\(^2\)

\[
\exists f_x, f_y. \forall x, y, \omega_1. (\omega_0 = 0 \land \varphi) \Rightarrow (\neg y \land \omega_3 \leq p)
\]

Above, \( \varphi \) is a set of conjuncts, each encoding the semantics of one statement in \( \tau_1 \):

\[
\varphi = \begin{cases} 
   f_x = 1 \Rightarrow x \land \omega_1 = \omega_0 + 0.5 \\
   f_x = 2 \Rightarrow \neg x \land \omega_1 = \omega_0 + 0.5 \\
   f_x = 3 \Rightarrow \omega_1 = \omega_0 \\
\end{cases} \land (x \land \omega_2 = \omega_1) \land \begin{cases} 
   f_y = 1 \Rightarrow y \land \omega_3 = \omega_2 + 1 - p \\
   f_y = 2 \Rightarrow \neg y \land \omega_3 = \omega_2 + p \\
   f_y = 3 \Rightarrow \omega_3 = \omega_2 \\
\end{cases}
\]

Let us explain how the encoding models the program \( P \). The variables \( \omega_j \) are fresh real-valued variables that represent the probability of failure along the path—\( \omega_0 \), the initial probability at node \( n \), is constrained to 0. The right-hand side of the implication in Eq. (1) encodes the postcondition \( \neg y \) and the upper bound on the failure probability \( \omega_3 \leq p \).

The more interesting parts of the encoding are the existentially quantified variables \( f_x, f_y \), which appear in \( \varphi \); we assume that \( f_x, f_y \in \{1,2,3\} \). These are used to select an axiomatization for each sampling statement. Synthesizing the right values for \( f_x \) and \( f_y \) allows us to show that Eq. (1) is valid, and therefore prove correctness of \( \tau_1 \). For instance, if \( f_x \) is set to 1, then \( x \sim bern(0.5) \) is encoded as an assignment statement \( x \leftarrow true \) with an accumulated failure probability of 0.5, since \( x \) is not \( true \) with a probability of 0.5; if \( f_x \) is set to 3, then \( x \) is treated as a non-deterministic Boolean, incurring no probability of failure.

---

1. This argument is an instance of a more general proof technique over sets of traces represented as automata; we will later formalize this idea as merging two automata.
2. We have simplified some aspects of the encoding here; §6 provides a formal treatment.
It is not hard to check that any proof of validity of Eq. (1) must set \( f_x = 3 \) and \( f_y = 2 \), as otherwise we cannot establish the postcondition, \( \neg y \), or the upper bound on failure, \( p \). In general, we treat \( f_x \) and \( f_y \) as uninterpreted functions whose arguments are program inputs, so that the choice of axiomatization may depend on the program state (§ 6 presents the general form).

**Labels via Craig Interpolation.** Suppose that we have proved validity of Eq. (1) and discovered that setting \( f_x = 3 \) and \( f_y = 2 \) yields a satisfiable formula. Plugging these values into Eq. (1) and negating the postcondition, we arrive at the following unsatisfiable formula:

\[
\omega_0 = 0 \land \omega_1 = \omega_0 \land x \land \omega_2 = \omega_1 \land \neg y \land \omega_3 = \omega_2 + p \land (y \lor \omega_3 > p)
\]

In first-order logic, it is known that if \( A \land B \) is unsatisfiable, then there is a formula \( I \) over the shared vocabulary of \( A \) and \( B \) such that \( A \Rightarrow I \) and \( I \Rightarrow \neg B \) are valid. \( I \) is called a Craig interpolant.

Intuitively, an interpolant overapproximates \( A \) while maintaining unsatisfiability with \( B \); this overapproximation can be seen as trying to generalize the assertions as much as possible. In our unsatisfiable formula above, we can compute a sequence of interpolants by splitting the formula into \( A \) and \( B \) segments after every statement’s encoding.\(^3\) The resulting interpolants compactly encode the two labels on traces, the sets of states and probabilities of failure. E.g., consider the split:

\[
A \triangleq \omega_0 = 0 \land \omega_1 = \omega_0 \land x \land \omega_2 = \omega_1 \land \neg y \land \omega_3 = \omega_2 + p \land (y \lor \omega_3 > p)
\]

\[
B \triangleq \neg y \land \omega_3 = \omega_2 + p \land (y \lor \omega_3 > p)
\]

A possible interpolant for \( A \land B \) is \( I \triangleq \omega_2 = 0 \). This indicates that any program state is reachable at node \( th \) (since program variables are unconstrained in \( I \)) with a probability of failure 0. The interpolant condition ensures that \( I \) can only mention \( \omega_2 \), the only variable shared by \( A \) and \( B \).

### 2.3 Illustrative Example: Handling Loops

We now consider a more complex example with loops, \( \text{ex2} \) in Fig. 4. \( \text{ex2} \) is a simplified sketch of mechanisms from differential privacy [Dwork et al. 2006], which carefully add random noise to query results before disclosing them. The program \( \text{ex2} \) takes an array of integers \( q \) of length \( n \), and constructs an array \( a \) whose values are noisy versions of those in \( q \). Specifically, for each element \( q[i] \), \( a[i] \) is noise drawn from the Laplace distribution with mean \( q[i] \) and scale \( 1/\epsilon \), where \( \epsilon > 0 \) is a real-valued input to the program. (All primitive distributions are defined in § 3.)

Our goal is to prove the accuracy property \( r_{p,n} \{ \text{true} \} \text{ex2}(q,n,\epsilon) \{ \varphi_{\text{post}} \} \), where the post-condition is defined to be

\[
\varphi_{\text{post}} \triangleq \forall j \in [0,n]. |a[j] - q[j]| \leq \frac{1}{\epsilon \log \left( \frac{1}{p} \right)}.
\]

In other words, for any \( p \in (0,1) \), we want to verify that the difference between \( a[j] \) and \( q[j] \) is bounded by a function of \( \epsilon \) and \( p \). Observe that \( \varphi_{\text{post}} \) involves input parameters \( q,n,\epsilon, \) and \( p \), but \( p \) does not appear in the program—the accuracy property is a parameterized family of properties. From our postcondition, we see that we can guarantee tighter bounds on the error—the difference between the exact answer \( q[j] \) and the noisy answer \( a[j] \)—if we are willing to allow this property to be violated with larger probability \( p \cdot n \). This style of postcondition is common for many randomized algorithms, capturing the relationship between accuracy—how far the results are from the exact values—and probability of failure, or how often the target property will not hold.

---

\(^3\)Equivalently, we can encode the problem as solving recursion-free Horn clauses [Rümmer et al. 2013].
Trace Generalization. The control-flow automaton representation of ex2 is shown in the box in Fig. 5. While the total number of loop iterations is at most the input parameter \( n \) in the original program, the automaton abstraction overapproximates these program behaviors with an infinite number of traces due to the loop. Therefore, unlike our first example, we cannot construct a proof for every trace individually. Our technique proceeds by picking traces, proving them correct, and generalizing them into automata representing infinite sets.

Let us first consider trace \( \tau_1 \) in Fig. 5; this trace does not enter the loop. The trace is easily shown to be correct since not entering the loop implies that \( n < 0 \), vacuously implying \( \varphi_{\text{post}} \) with failure probability 0. More interesting is trace \( \tau_2 \) in Fig. 5, which executes the loop body once and exits. The formula \( \varphi \) in the annotation is defined as follows:

\[
\varphi \triangleq \forall j \in [0, i). |a[j] - q[j]| \leq \frac{1}{\varepsilon} \log \left( \frac{1}{p} \right)
\]

Notice the probability of failure is \( p \cdot i \) on nodes \( w, w1, w2, \) and \( w' \). After loop exit, using the exit condition, we conclude that the probability of failure is \( p \cdot n \). Informally, these labels capture the fact that the failure probability depends on how many times we have executed the loop, which is tracked in the counter \( i \).

Our algorithm discovers that the labels are inductive: no matter how many times we execute the loop, the probability of failing to satisfy \( \varphi \land i < n \) at loop entry is \( p \cdot i \). Therefore, the algorithm generalizes this trace into an infinite set of traces by adding an edge from node \( w' \) to \( w1 \) with the statement \( i < n \). With this additional edge in place, we now have an automaton representing all traces that go through the loop at least once. The total failure probability of those traces is the label under node \( a \cdot c: p \cdot n \). Combined with trace \( \tau_1 \), we have covered all the traces of ex2, proving that the total probability of failure is \( p \cdot n + 0 = p \cdot n \) as desired.

Selecting Axioms for the Laplace Distribution. The sampling statement \( a[i] \sim \text{lap}(\ldots) \) in \( \tau_2 \) is encoded by the following logical formula:

\[
|a[i] - q[i]| \leq \frac{1}{\varepsilon} \log \left( \frac{1}{f_a(i,p,n)} \right) \land \omega_3 = \omega_2 + f_a(i,p,n)
\]

The left conjunct specifies that we can assume that the difference between \( a[i] \) and \( q[i] \) is at most \( \frac{1}{\varepsilon} \log \left( \frac{1}{f_a(i,p,n)} \right) \); the right conjunct specifies that this assumption fails with a probability of \( f_a(i,p,n) \). We treat \( f_a \) as an uninterpreted function with range \((0,1]\), so that there are infinitely many possible interpretations of \( f_a \) corresponding to different failure probability/accuracy tradeoffs for the Laplace distribution. To get the annotation proving correctness of \( \tau_2 \) in Fig. 5, our technique synthesizes

\[In practice, we treat non-linear arithmetic operations and transcendentals (e.g., log) as uninterpreted functions and use the theorem-enumeration technique recently proposed by Srikanth et al. [2017] to axiomatize them.\]
the interpretation \( f_s(i, p, n) = p \). With this choice, our analysis accumulates a probability of failure of \( p \) for every loop iteration, ending up with a total probability of failure of \( p \cdot n \).

### 3 Preliminaries

In this section, we formalize our program model and accuracy specifications.

#### 3.1 Program Model and Semantics

**Probability Distributions.** To model probabilistic computation mathematically, we use probability sub-distributions. A function \( \mu : C \to [0, 1] \) defines a discrete sub-distribution over a set \( C \) if it is non-zero for at most countably many elements in \( C \), and \( \sum_{c \in C} \mu(c) \leq 1 \); we will abbreviate discrete sub-distribution as distribution throughout this paper. We will often write \( \mu(C') \) for a subset \( C' \subseteq C \) to mean \( \sum_{c \in C'} \mu(c) \). We write \( \text{dist}(C) \) for the set of all distributions over \( C \). The support of a distribution \( \mu \) is defined as \( \text{supp}(\mu) \triangleq \{ c \in C \mid \mu(c) > 0 \} \).

We focus on discrete sub-distributions to keep measure-theory overhead to a minimum. As a consequence, we only allow programs to sample from primitive discrete distributions. Supporting continuous primitive distributions (e.g., the Gaussian distribution) would not introduce any difficulties beyond requiring a more technically involved definition of the program semantics.

**Program Expressions.** We fix a set of variables \( V \) that appear in the program. A program state \( s \) is a map assigning every variable \( v \in V \) to a value. We will use \( S \) to denote the set of all possible states. Given variable \( v \), we use \( s(v) \) to denote the value of \( v \) in state \( s \). Given constant \( c \), we use \( s[v \mapsto c] \) to denote the state \( s \) with variable \( v \) mapped to \( c \). The semantics of an expression \( e \) is a function \( s : S \to D \) from a state to an element of some type \( D \). For instance, the expression \( x + y \) in state \( s \) is interpreted as \( [x + y](s) = s(x) + s(y) \). We will often abbreviate \([e](s)\) by \( s(e)\).

**Distribution Expressions.** A distribution expression \( d \) is interpreted as a distribution family \( [d] : S \to \text{dist}(D) \), mapping a state in \( S \) to a distribution over \( D \) with countable support. Our framework can naturally handle any distribution expression that can be interpreted as a discrete distribution. For concreteness, we will consider the four primitive distributions in Table 1.

Consider the Bernoulli distribution expression, \( \text{bern}(e) \). Given a state \( s \), semantically \( \text{bern}(e) \) is the distribution \( \mu \in \text{dist}(\mathbb{B}) \) where \( \mu(\text{true}) = s(e) \) and \( \mu(\text{false}) = 1 - s(e) \). Similarly, the uniform distribution expression \( \text{unif}(e) \), where \( e \) encodes to a finite set, is interpreted as the distribution assigning equal probability to every element in \( s(e) \).

We also use the (discrete) Laplace distribution, a common primitive distribution in the theory of differential privacy. For a state \( s \), the distribution expression \( \text{lap}(e_1, e_2) \) is semantically the discrete Laplace distribution with \( \text{mean} s(e_1) \) and \( \text{scale} s(e_2) \): for every integer \( c \in \mathbb{Z} \), it assigns a probability proportional to \( \exp\left(-\frac{|c-s(e_1)|}{s(e_2)}\right) \). The (discrete) exponential distribution expression \( \exp(e_1, e_2) \) is similar, but only assigning positive probability to integers above the shift \( s(e_1) \).

We implicitly assume that arguments of distribution expressions are well-typed and valid.

---

**Table 1. Distribution expressions and their semantics**

<table>
<thead>
<tr>
<th>Name</th>
<th>Dist. expr. ( d )</th>
<th>Parameters</th>
<th>Semantics ( s(d) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli</td>
<td>( \text{bern}(e) )</td>
<td>( e \in [0, 1] )</td>
<td>( \mu(\text{true}) = s(e) ) and ( \mu(\text{false}) = 1 - s(e) )</td>
</tr>
<tr>
<td>Uniform</td>
<td>( \text{unif}(e) )</td>
<td>( e ) is a finite set</td>
<td>( \mu(c) = 1/</td>
</tr>
<tr>
<td>Laplace</td>
<td>( \text{lap}(e_1, e_2) )</td>
<td>mean ( e_1 \in \mathbb{Z} ); scale ( e_2 \in \mathbb{R}_{\geq 0} )</td>
<td>( \mu(c) \propto \exp\left(-\frac{</td>
</tr>
<tr>
<td>Exponential</td>
<td>( \text{exp}(e_1, e_2) )</td>
<td>shift ( e_1 \in \mathbb{Z} ); scale ( e_2 \in \mathbb{R}_{\geq 0} )</td>
<td>( \mu(c) \propto \exp\left(-\frac{c-s(e_1)}{s(e_2)}\right) ), for ( c \in \mathbb{Z} ) and ( c \geq s(e_1) )</td>
</tr>
</tbody>
</table>
We can encode the set of possible traces of a program where each term $A$ Trace Semantics.

1:10 Calvin Smith, Justin Hsu, and Aws Albarghouthi


Programs, Statements, and Traces. Our verification technique will target programs written in a probabilistic, imperative language. The basic statements are drawn from a set $\Sigma$:

- **Assignment** statements $v \leftarrow e$, where $e$ is an expression over $V$, e.g., $v_1 \leftarrow v_2$.
- **Sampling** statements $v \sim d$, where $d$ is a distribution expression.
- **Assume** statements assume($b$), where $b$ is a Boolean expression over $V$.

A trace $T$ is a finite sequence of statements $s_1; \cdots ; s_n$, and a program $P$ is interpreted as a (possibly infinite) set of traces $L(P)$. We include full details of the programming language in the extended version [Smith et al. 2018]; the interpretation is standard, using assume statements to model typical control-flow constructs. For instance, a conditional statement if $b$ then $T_1$ else $T_2$ can be modeled as the pair of traces assume($b$); $T_1$ and assume($\neg b$); $T_2$. By construction, traces in $L(P)$ are semantically disjoint—no trace in $L(P)$ is a prefix of (or equal to) any other trace in $L(P)$, and the first differing statements between any two traces are of the form assume($b$) and assume($\neg b$).

Trace Semantics. We interpret a trace $T$ as a function $[T] : S \rightarrow dist(S)$ from input states to distributions over output states. To define this semantics formally, we need two standard constructions on distributions. The map $\text{unit} : D \rightarrow dist(D)$ maps $a \in D$ to the Dirac distribution $\delta_a$ at $a$, i.e., the distribution that returns 1 at $a$ and 0 otherwise. The map $\text{bind} : dist(D_1) \rightarrow (D_1 \rightarrow dist(D_2)) \rightarrow dist(D_2)$ combines probabilistic computations in sequence: $\text{bind}(\mu, f)(a_2) = \sum_{a_1 \in D_1} \mu(a_1) \cdot f(a_1)(a_2)$. These maps are the usual unit and bind for the (sub-)distribution monad.

3.2 Programs as Automata

We can encode the set of possible traces of a program $P$ as a regular language $L(P)$ represented by all paths through its control-flow graph. We begin with a general definition of automata over program statements, and then show how we represent programs as automata.

Automata over Statements. A finite-state automaton over statements $A$ is a graph $(Q, \delta)$, where

- $Q$ is a finite set of nodes.
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, where $\Sigma$ is basic statements.
- $q^{\text{in}}, q^{\text{ac}} \in Q$ are special nodes called the initial and accepting nodes, respectively.

We will use $q_i \xrightarrow{s_i} q_j$ to denote that $(q_i, s, q_j) \in \delta$. We write $L(A)$ for the language of traces accepted by $A$, where a trace $s_1, \ldots, s_n$ is accepted iff $[q^{\text{in}} \xrightarrow{s_1} q_1, q_1 \xrightarrow{s_2} q_2, \ldots, q_{n-1} \xrightarrow{s_n} q^{\text{ac}}] \subseteq \delta$. It will sometimes be useful to use multiple automata to model the traces in a single program. We will use $L(\mathcal{A})$ to denote the union of all languages accepted by a set of automata $\mathcal{A}$, i.e., $\bigcup_{A \in \mathcal{A}} L(A)$.
We assume that all nodes \( q \in Q \) can reach the accepting node \( q^{ac} \) via the transition relation \( \delta \), and that there are no transitions starting from \( q^{ac} \). We also assume that automata model well-formed control flow, i.e., (i) all nodes \( q_i \in Q \) have at most two outgoing transitions and (ii) if \( q_i \xrightarrow{s} q_j \) and \( q_i \xrightarrow{\beta} q_k \) for \( j \neq k \), then \( s_1, s_2 \) are of the form \( \text{assume}(b_1) \) and \( \text{assume}(b_2) \), such that \( b_1 \equiv \neg b_2 \).

**From Program Traces to Automata** We will identify a program with an automaton representing its control-flow graph (CFG). A program \( P \) is of the form \((L, \delta)\), where the nodes \( L \) of the automaton denote the set of program locations (e.g., line numbers). The special nodes \( t^{in}, t^{ac} \in L \) model the first and last lines of the program. To ensure there is no control-flow non-determinism, we assume that for any \( \ell_i \xrightarrow{\text{assume}(b)} \ell_j \), there is a transition \( \ell_i \xrightarrow{\text{assume}(\neg b)} \ell_k \).

We use \( V^{in} \subseteq V \) to denote the set of input variables, which are not modified by the program. We will also use \( V^{det} \subseteq V \) to denote the set of program variables whose values are assigned deterministically, i.e., not affected by probabilistic choice—by definition, \( V^{in} \subseteq V^{det} \). (We may not be able to determine \( V^{det} \) exactly in practice, but we can under-approximate it via a simple static analysis.)

### 3.3 Probabilistic Accuracy Properties

We will define specifications using the Hoare-style statement

\[
\vdash_{\beta} \{ \varphi_{pre} \} \tau \{ \varphi_{post} \}
\]

where the precondition \( \varphi_{pre} \subseteq S \) and postcondition \( \varphi_{post} \subseteq S \) are sets of program states, and the failure probability \( \beta \) is a \([0, 1]\)-valued function over input variables \( V^{in} \). For simplicity, we will treat \( \beta \) as an expression over \( V^{in} \)—e.g., 0 or \( p \cdot n \) in § 2.3—and use \( s(\beta) \) to denote the value of \( \beta \) in state \( s \).

We say that \( \vdash_{\beta} \{ \varphi_{pre} \} \tau \{ \varphi_{post} \} \) is valid if for any state \( s \in \varphi_{pre} \), we have \( \mu(\varphi_{post}) \leq s(\beta) \), where \( \mu = [\tau](s) \) and \( \varphi_{post} = S \setminus \varphi_{post} \). In other words, the probability that the trace starts in \( \varphi_{pre} \) and does not end up in \( \varphi_{post} \) is upper bounded by \( \beta \). We extend this notation to programs \( P \) in the natural way, writing \( \vdash_{\beta} \{ \varphi_{pre} \} P \{ \varphi_{post} \} \) if for any input state \( s \in \varphi_{pre} \), the output distribution \( \mu = [P](s) \) satisfies the bound \( \mu(\varphi_{post}) \leq s(\beta) \).

### 4 TRACE ABSTRACTION MODULO PROBABILITY

With the preliminaries out of the way, we begin to introduce a version of trace abstraction for probabilistic programs and show how to use it to prove accuracy specifications. Given a program \( P \), suppose we want to establish the following accuracy specification: \( \vdash_{\beta} \{ \varphi_{pre} \} P \{ \varphi_{post} \} \). We will over-approximate the traces of \( P \) with a set of automata \( \mathcal{A} \) and analyze each automaton separately; this way, we can focus on smaller groups of possible traces. If we can show that the probability \( \varphi_{post} \) does not hold across all automata is at most \( \beta \), this implies the accuracy specification. We formalize this argument in the following proof rule. (We defer all proofs to the extended version [Smith et al. 2018].)

**Theorem 4.1 (General proof rule).** The specification \( \vdash_{\beta} \{ \varphi_{pre} \} P \{ \varphi_{post} \} \) is valid if there exists a set of automata \( \mathcal{A} \) such that

\[
\mathcal{L}(P) \subseteq \mathcal{L} \quad \text{(Trace inclusion)}
\]

\[
\forall s \in \varphi_{pre}, \sum_{\tau \in \mathcal{L}(\mathcal{A})} [\tau](s)(\varphi_{post}) \leq s(\beta) \quad \text{(Failure probability upper bound)}
\]
enrich the automata with additional information on each state; local properties of these labeled automata will then imply the failure probability upper bound.

**Enriching Automata with Labels.** We work with automata where every node is labeled with a predicate on states (equivalently, a set of states), and a function representing the failure probability—we call such automata failure automata. The rough intuition is that at each node \( q \), the predicate label represents a program invariant that holds on all traces reaching \( q \) from the beginning of the program, except with probability given by the failure probability label.

**Definition 4.2 (Failure automata).** A failure automaton \( A = (Q, \delta, \lambda, \kappa) \) is an automaton \( (Q, \delta) \) with two labeling functions, \( \lambda \) and \( \kappa \), where

- \( \lambda \) maps every node \( q \in Q \) to a set of states, and
- \( \kappa \) maps every node \( q \in Q \) to a \([0,1]\)-valued function over \( \mathcal{V}^{\det} \).

We say that \( A \) is well-labeled iff the following conditions hold:

1. \( \kappa(q^n) = 0 \) and \( \kappa(q^{ac}) \) is a \([0,1]\)-valued function over the input variables \( V^{in} \subseteq V^{\det} \), and
2. for every transition \( q_i \xrightarrow{A} q_j \), the statement \( \vdash_{wp} (\kappa(q_j), x) \rightarrow_{\kappa(q_i)} \{ \lambda(q_j) \} \not\in \{ \lambda(q_i) \} \), is valid where \( wp \) is a weakest-precondition operation over failure-probabilities: \( wp(e, x) = e \) for assume and sampling statements, and \( wp(e_1, v \leftarrow e_2) = e_1[\{v\} \mapsto e_2] \).

The two conditions ensure that if we take any trace \( \tau \in \mathcal{L}(A) \), then \( \vdash_{\kappa(q^{ac})} \{ \lambda(q^n) \} \not\in \{ \lambda(q^{ac}) \} \) is valid. Point (2) ensures that failure probability accumulates additively as we move along the trace, starting from being 0 at \( q^n \), as stipulated by point (1). Crucially, both points are local conditions: they can be easily checked given a failure automaton. However, coming up with well-labeled automata for a given program is not at all trivial—we return to this question in the next two sections.

**Example 4.3.** Recall our example from § 2.2, illustrated in Fig. 3. The lower part of Fig. 3 shows a failure automaton named \( \tau'_3 \) with \( \lambda \) and \( \kappa \) shown above and below the nodes, respectively. Notice that the initial node \( n_1 \) is labeled with \( \lambda(n_1) \triangleq true \) and \( \kappa(n_1) \triangleq 0 \). Focusing on the edge from node \( n_3 \), the labeling at \( ac \) satisfies condition (2) for well-labeledness in Definition 4.2. The condition says that the following statement must be valid: \( \vdash_{\kappa(q^{ac})} \{ \lambda(n_1) \} \not\in \{ \lambda(q^{ac}) \} \). The failure probability \( p \) is the simplification of the expression \( wp(0.5 + p, y \sim bern(p)) - 0.5 \). The statement is valid since \( y \) is true with probability \( p \) after executing \( y \sim bern(p) \).

The following theorem establishes soundness of annotations on well-labeled automata. Specifically, the failure probability label on \( q^{ac} \)—namely, \( \kappa(q^{ac}) \)—is an upper bound on the probability that executions through \( A \) do not end up in a state in \( \lambda(q^n) \).

**Theorem 4.4 (Well-labeled automata soundness).** Let \( A \) be a well-labeled failure automaton. Then, for every \( s \in \lambda(q^n) \) and \( \mu = [\tau](s) \), we have \( \sum_{\tau \in \mathcal{L}(A)} \mu(\lambda(q^{ac})) \leq s(\kappa(q^{ac})) \).

**Proofs from Well-labeled Automata.** Now that we have established soundness of well-labeled automata, we refine our original proof rule (Theorem 4.1) using failure automata. The following theorem demonstrates how to establish correctness using a set of failure automata.
Theorem 4.5 (Proof rule with failure automata). The statement \( \vdash_{\beta} \varphi_{\text{pre}} \ P \ \varphi_{\text{post}} \) is valid if there exist well-labeled automata \( \mathcal{A} = \{A_1, \ldots, A_n\} \) such that the following conditions hold:

\[
\begin{align*}
\mathcal{L}(P) &\subseteq \mathcal{L}(\mathcal{A}) \\
\forall i \in [1, n], \varphi_{\text{pre}} &\subseteq \lambda_i(q_i^{\text{in}}) \\
\forall i \in [1, n], \lambda_i(q_i^{\text{ac}}) &\subseteq \varphi_{\text{post}} \\
\forall s \in \varphi_{\text{pre}}, \sum_{i=1}^{n} s(\kappa_i(q_i^{\text{ac}})) &\leq s(\beta)
\end{align*}
\]

The trace, precondition, and postcondition inclusion conditions are the same as in trace abstraction for non-probabilistic programs. The failure probability upper bound condition ensures that the overapproximation of failure probability resulting from abstraction does not exceed \( \beta \). Notice that precondition and postcondition inclusion checks can be performed using an SMT solver, assuming labels are encoded in a supported first-order theory. Similarly, the failure probability upper bound condition involves summing up the labels on the accepting nodes of all failure automata, allowing us to perform the check with an SMT solver.

Example 4.6. Recall the example program \( \text{ex2} \) from § 2.3, illustrated in Fig. 5. The two automata, denoted \( r_1 \) and \( r_2 \) in Fig. 5, are well-labeled. The automata cover all program traces. The initial nodes, denoted \( i^n \), have the labels \( \lambda \) as \text{true}, therefore satisfying the precondition inclusion condition. The accepting nodes, denoted \( \text{ac} \), both imply the postcondition, \( \varphi_{\text{post}} \). Finally, the sum of the failure probabilities on accepting nodes is \( 0 + p \cdot n \leq p \cdot n \), satisfying the failure probability condition.

5 CONSTRUCTING TRACE ABSTRACTIONS

Theorem 4.5 reduces checking accuracy properties to finding a set of well-labeled automata. Our algorithm for automating this proof rule is technically complex, and spans the following two sections. Here, we will present the algorithm and prove soundness, assuming a procedure for...
well-labeling single traces; we will detail this last piece in § 6. Then, we compare our algorithm with two existing techniques: the union bound logic AHI, and standard trace abstraction.

5.1 Algorithm Overview

Our algorithm maintains a set \( \{ A_i \} \) of well-labeled failure automata modeling some of the program traces, and repeatedly finds traces \( \tau \in \mathcal{L}(P) \) that are not in \( \{ A_i \} \). If a trace can be well-labeled, it is converted into a well-labeled automaton \( A_i \) proving that \( \tau \) \( \{ \phi_{\text{pre}} \} \tau \{ \phi_{\text{post}} \} \) and added to the current automaton set. Throughout, the algorithm may simplify or transform the automaton set by merging automata together and generalizing automata by adding new edges. The process terminates successfully if the set of failure automata \( \{ A_i \} \) satisfies the conditions in Theorem 4.5.

The input to the algorithm is a program \( P \), a pre- and post-condition \( \phi_{\text{pre}} \) and \( \phi_{\text{post}} \), and a target failure probability \( \beta \), a function over the input variables of the program. The entire algorithm is presented in Fig. 7 as a set of non-deterministic guarded rules. The algorithm preserves the invariant that the set of automata \( \mathcal{A} \) are well-labeled. We briefly consider each rule in turn.

Initialization. The rule INIT is the only rule with no premises and serves as the initialization rule. Not surprisingly, the set of failure automata \( \mathcal{A} \) is initially empty.

Trace Sampling. The rule TRACE picks a trace \( \tau \) that is in the program \( P \) but not covered by the set of automata \( \mathcal{A} \). It then uses the function label to construct a well-labeled automaton \( A_i \), implying that \( \tau \) \( \{ \phi_{\text{pre}} \} \tau \{ \phi_{\text{post}} \} \). We will detail the label operation in § 6; for now, we just note that label may fail, in which case the rule TRACE does not fire and the algorithm tries a different trace.

Generalizing Automata. The rule GENERALIZE expands the language \( \mathcal{L}(\mathcal{A}) \) by adding new edges to an automaton \( A \in \mathcal{A} \). When the new edges form loops, this rule can be seen as generalizing from automata modeling finite unrollings of looping statements to automata overapproximating loops. The side-conditions ensure that this transformation preserves well-labeledness.

Merging Automata. The rule MERGE combines automata whose traces are mutually exclusive, allowing us to take the maximum failure probability instead of the sum. Intuitively, automata that begin with the same prefix of statements before making mutually exclusive assumptions—say, assume(\( b \)) and assume(\( \neg b \))—can have their prefixes merged together if they have equivalent labels. This operation can be seen as constructing an automaton combining two branches of a conditional. Concretely, the operator \( \boxtimes \) takes two automata, \( A_1 \) and \( A_2 \), and returns a new automaton that accepts the union of the traces. We formalize \( \boxtimes \) and its preconditions below:

\[
\text{Definition 5.1. We assume the two automata } A_1, A_2 \text{ of the form } A_i = (Q_i, \delta_i, \lambda_i, \kappa_i) \text{ with the initial and final nodes } q_{1i}^{\text{in}}, q_{1i}^{\text{ac}}. \text{ Suppose there is a prefix of statements } s_1, \ldots, s_n \text{ such that}
\]

1. every path from \( q_{1i}^{\text{in}} \) to \( q_{1i}^{\text{ac}} \) is of the form \( q_{1i}^{\text{in}} \xrightarrow{s_1} q_{1,1} \xrightarrow{s_2} q_{1,2} \cdots q_{1,n} \xrightarrow{\text{assume}(b)} q_{1,n+1} \cdots q_{1i}^{\text{ac}} \).
2. every path from \( q_{2i}^{\text{in}} \) to \( q_{2i}^{\text{ac}} \) is of the form \( q_{2i}^{\text{in}} \xrightarrow{s_1} q_{2,1} \xrightarrow{s_2} q_{2,2} \cdots q_{2,n} \xrightarrow{\text{assume}(\neg b)} q_{2,n+1} \cdots q_{2i}^{\text{ac}} \).
3. each prefix node \( q \in \{ q_{1i}^{\text{in}}, q_{1,1}, \ldots, q_{1,n} \} \) has equivalent labels (\( \lambda \) and \( \kappa \)) to its corresponding node in \( \{ q_{2i}^{\text{in}}, q_{2,1}, \ldots, q_{2,n} \} \).

Then, \( A_1 \boxtimes A_2 \) yields a failure automaton \( A = (Q, \delta, \lambda, \kappa) \) with

- \( Q = Q_1 \cup \{ Q_2 \setminus \{ q_{2i}^{\text{in}}, q_{2,1}, \ldots, q_{2,n}, q_{2i}^{\text{ac}} \} \} \);
- \( \delta = \delta_1 \cup \delta_2 \cup \{ q_i \xrightarrow{s} q_{2i}^{\text{ac}} | q_i \xrightarrow{\neg s} q_{2i}^{\text{ac}} \in \delta_2 \} \), with all edges to/from undefined nodes removed;
- \( q^{\text{in}} = q_{1i}^{\text{in}} \) and \( q^{\text{ac}} = q_{1i}^{\text{ac}} \);
- \( \lambda \) agrees with \( \lambda_1 \) and \( \lambda_2 \), except that \( \lambda(q^{\text{ac}}) = \lambda(q_{1i}^{\text{ac}}) \cup \lambda(q_{2i}^{\text{ac}}) \); and
- \( \kappa \) agrees with \( \kappa_1 \) and \( \kappa_2 \), except that \( \kappa(q^{\text{ac}}) = \max(\kappa(q_{1i}^{\text{ac}}), \kappa(q_{2i}^{\text{ac}})) \).
More advanced extensions of this operation are also possible, for instance, also merging common post-fixes along with common prefixes, but we stick with this version for concreteness.

If two automata are well-labeled and the merge rule applies, then the resulting merged automaton is also well-labeled. It is, however, important to note condition (3) in Definition 5.1, which states that the shared prefix between the two automata must have the same labels on both automata. If that condition is violated, the result may not be well-labeled, as illustrated in the following example.

**Example 5.2.** Consider the two well-labeled single-trace automata $A_1$ and $A_2$ in Fig. 8(a), which model a conditional statement and share the prefix $x \sim \text{bern}(0.5)$. The annotations prove that both traces satisfy $\tau_{0.5}\{\text{true}\} A_i \{y > 0\}$. The operation $\wedge$ does not apply here, since the automata disagree on the label of the second node. However, suppose that we apply $\wedge$ nonetheless. This results in a final node with failure probability $\max(0.5, 0.5) = 0.5$. But this is not sound, since the probability of failing to achieve $y > 0$ is 1 when both traces are considered together, since both traces set $y$ to 0.

We also give an example of a sound application of merge.

**Example 5.3.** Consider the two well-labeled automata $\tau_1$ and $\tau_2$ from Fig. 3 in §2.2. They satisfy the conditions for $\wedge$. Fig. 8(b) shows the result of applying $\wedge$ to these two automata. Notice that the accepting node, denoted $ac$, has a label $\kappa(ac) = \max(p, 0.5p)$, which is equal to $p$.

**Lemma 5.4.** If $A_1, A_2$ are well-labeled and satisfy the $\wedge$ conditions, then $A = A_1 \wedge A_2$ is well-labeled.

**Termination.** Finally, the rule correct gives the termination condition for the algorithm, corresponding to the conditions from Theorem 4.5. Notice that precondition and postcondition inclusion hold by construction, since they were ensured by the labeling function label when the first trace in each automaton was added to the automaton set by rule trace.

**5.2 Theoretical Properties**

**Soundness.** As expected, the algorithm is sound.

**Theorem 5.5 (Soundness of Algorithm).** If correct applies, then $\tau_{\beta}\{\varphi_{pre}\} P\{\varphi_{post}\}$ is valid.

**In(completeness.** Our approach is generally incomplete. The incompleteness primarily results from the application of the union bound, which, in some programs, does not allow us to prove the tightest possible failure probabilities. E.g., consider $\tau_{0.75}\{\text{true}\} x \sim \text{bern}(0.5); y \sim \text{bern}(0.5) \{x \land y\}$. Any well-labeled automaton will upper bound the failure probability by 1, since we have no means of assuming independent sampling in both statements. This example can be handled by coalescing the two sampling statements into a single statement; however, the general issue arises in loops, too.

Nevertheless, we can compare the expressivity of our approach with two existing techniques: the union bound logic [Barthe et al. 2016b] and trace abstraction [Heizmann et al. 2009, 2013].
Union Bound Logic. The union bound logic [Barthe et al. 2016b] is an extension of Hoare logic with failure probabilities, where Hoare triples are analogous to our annotations \( \vdash_\beta \{ \varphi_{\text{pre}} \} P \{ \varphi_{\text{post}} \} \). Our notion of well-labeled automata can capture proofs in the union bound logic with the exception of a few points, and our algorithm can recover a precise class of well-labeled automata. We formalize this correspondence and prove a completeness result in the extended version [Smith et al. 2018].

Trace Abstraction. Our technique generalizes trace abstraction for non-probabilistic, single-procedure programs [Heizmann et al. 2009, 2013]. When given a non-probabilistic program \( P \) and Hoare triple \( \{ \varphi_{\text{pre}} \} P \{ \varphi_{\text{post}} \} \), we can construct trace-abstraction proofs by simply setting the failure probability upper bound to 0 in the specification. Consequently, the failure probability labels of nodes of all automata in \( \mathcal{A} \) must be 0 for the proof to hold. In this setting, the state labels (\( \lambda \)) are overapproximations of reachable states at a specific node, corresponding to the annotations of Floyd–Hoare automata defined by Heizmann et al. [2013].

6 LABELING INDIVIDUAL TRACES: PROOFS AND INTERPOLATION

In the algorithm we presented in Fig. 7, the key subroutine is the label operation for rule TR. Recall that given a single trace \( \tau \), pre- and post-conditions \( \varphi_{\text{pre}} \) and \( \varphi_{\text{post}} \), and failure probability \( \beta \), label attempts to construct a well-labeled automaton \( A_\tau \) for \( \tau \) proving \( \vdash_\beta \{ \varphi_{\text{pre}} \} \tau \{ \varphi_{\text{post}} \} \). We now show how to reduce this task to a constraint-solving problem. Our approach is inspired by lazy abstraction with interpolants [McMillan 2006], where the semantics of \( \tau \) are encoded as a formula in first-order logic to check if it falsifies a Hoare triple. If the trace does not falsify the triple, Craig interpolants are computed along the trace forming a Hoare-style annotation. However, our setting is richer: we need to (i) handle traces with probabilistic semantics and (ii) construct two kinds of annotations—sets of states and failure probability expressions. We demonstrate how to reduce this problem to Craig interpolation over a first-order theory, thus eliminating probabilistic reasoning. We summarize our approach below:

1. **Axiomatizing distributions:** We demonstrate how to encode \( \vdash_\beta \{ \varphi_{\text{pre}} \} \tau \{ \varphi_{\text{post}} \} \) as a logical formula. The key challenge is in encoding semantics of sampling statements. We address this challenge by observing that we can encode sampling statements by introducing appropriate logical axioms about the distributions. This results in a constraint-based synthesis problem of the form \( \exists f. \forall X. \varphi \), where discovering a function \( f \) amounts to finding an appropriate axiom for each sampling statement in order to establish correctness of the trace.

2. **Craig interpolation:** Once we have solved the synthesis problem by finding a solution for \( f \), we are left with a valid logical formula of the form \( \forall X. \varphi \), which we can use to compute interpolants using standard techniques. We demonstrate that these interpolants can be converted to a well-labeling of \( A_\tau \).

### Table 2. Example families of distribution axioms (\( \nu \) does not occur in dist. expr.)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Assumption ( \varphi^{ax} )</th>
<th>Upperbound ( \varphi^{ub} )</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bernoulli: ( \nu \sim \text{bern}(\nu') )</td>
<td>( f(V^{in}) = 1 \land \nu \lor f(V^{in}) = 2 \land \neg\nu )</td>
<td>( \nu' ) if ( f(V^{in}) = 1 ) ( 1 - \nu' ) if ( f(V^{in}) = 2 ) otherwise</td>
<td>( f(V^{in}) \in {1, 2, 3} )</td>
</tr>
<tr>
<td>Uniform: ( \nu \sim \text{unif}(\nu') )</td>
<td>( \nu \in f(V^{in}) )</td>
<td>(</td>
<td>f(V^{in})</td>
</tr>
<tr>
<td>Laplace: ( \nu \sim \text{lap}(\nu_1, \nu_2) )</td>
<td>(</td>
<td>\nu - \nu_1</td>
<td>&gt; \nu_2 \log \left( \frac{1}{f(V^{in})} \right) ) ( f(V^{in}) )</td>
</tr>
<tr>
<td>Exponential: ( \nu \sim \exp(\nu_1, \nu_2) )</td>
<td>( \nu &lt; \nu_1 \lor \nu - \nu_1 &gt; \nu_2 \log \left( \frac{2}{f(V^{in})} \right) ) ( f(V^{in}) )</td>
<td>( f(V^{in}) \in (0, 1] )</td>
<td></td>
</tr>
</tbody>
</table>
6.1 Proofs via Distribution Axiomatization

We now describe how we can check validity of the specification $\tau_\beta \{\varphi_{\text{pre}}\} \Gamma_1; \ldots; \Gamma_n \{\varphi_{\text{post}}\}$. Our approach is analogous to logical encodings of program paths in verification of non-probabilistic programs; there, each statement $\Gamma_i$ is encoded as a formula $\varphi_{\Gamma_i}$ in some appropriate first-order theory, e.g., the theories of linear arithmetic or arrays. Novel to our setting, we use distribution axioms to approximate the semantics of sampling statements in a first-order theory.

Logical Theory. We assume that deterministic program expressions correspond to a first-order theory, like linear arithmetic. Given a formula $\varphi$, a model $M$ of $\varphi$, denoted $M \models \varphi$, is a valuation of its free variables $fV(\varphi)$ satisfying the formula—e.g., $M \models x + y > 0$ where $M = \{x \mapsto 0, y \mapsto 1\}$. We use $M(\varphi)$ to denote $\varphi$ with all free variables replaced by their interpretation in $M$. A formula $\varphi$ is satisfiable if there exists $M$ such that $M \models \varphi$; a formula is valid if $M \models \varphi$ for all models $M$.

Distribution Axioms. Given a sampling statement $v \sim d$, an axiom is of the form

$$\Pr_{v \sim d}[\varphi^{ax}] \leq e^{ub}$$

where $e^{ub}$ is a $[0,1]$-valued expression over $V$ and $\varphi^{ax}$ is a formula over $V$. The axiom must be true for all possible valuations of the program variables $V \setminus \{v\}$. We will use the axioms as follows: When encoding the effect of a sampling statement $v \sim d$, we can assume that $\neg \varphi^{ax}$ is true, with a failure probability of at most $e^{ub}$. This allows us to sidestep probabilistic reasoning and encode program semantics in our first-order theory.

Since axioms are approximations of primitive distributions, there are many possible axioms for any given distribution. In some cases, axioms may be parameterized, e.g., by the failure probability. We call parameterized axioms axiom families; Table 2 collects example axiom families for the distributions in § 3.1.

Definition 6.1 (Laplace axiom family). Recall that the (discrete) Laplace distribution expression $\text{lap}(v_1, v_2)$ is parameterized by two parameters, the mean $v_1 \in \mathbb{Z}$ and the scale $v_2 \in \mathbb{R}$. Sampling from $\text{lap}(v_1, v_2)$ returns an integer $v$ with probability proportional to $\exp(-|v - v_1|/v_2)$. The Laplace distribution supports the following family of axioms, parameterized by a $(0,1)$-valued function $f$:

$$\Pr_{v \sim \text{lap}(v_1, v_2)} \left[ |v - v_2| > \frac{1}{v_1} \log \left( \frac{1}{f(V^{in})} \right) \right] \leq f(V^{in})$$

Different instantiations of $f$ yield different axioms.

The exponential distribution’s axiom family is similar; note that $\exp(v_1, v_2)$ has zero probability of returning elements smaller than $v_1$, and this information is incorporated into the axiom. The Bernoulli distribution’s family is parameterized by a function $f(V^{in})$ which decides whether to assume $v$ is true, false or treat it non-deterministically. The uniform distribution’s axiom family is parameterized by a function $f(V^{in})$ returning a subset of the set defined by $v'$.

Example 6.2. Recall trace $t_2$ (from program ex2) in § 2.3 and Fig. 5, which contains the statement $a[i] \sim \text{lap}(q[i], 1/\epsilon)$. To prove correctness of $t_2$, we instantiated the Laplace axiom family with $f(V^{in}) = p$ where $p \in V^{in}$, yielding the axiom $\Pr_{a[i] \sim \text{lap}(q[i], 1/\epsilon)} \left[ |a[i] - q[i]| > \frac{1}{\epsilon} \log \left( \frac{1}{p} \right) \right] \leq p$.

Theorem 6.3. Each axiom in Table 2 is sound: given any input state $s$ and well-typed distribution expression $d$, the probability that $\varphi^{ax}$ holds in $s(d)$ is at most $s(e^{ub})$.

Logical Encoding. We now present our encoding for checking $\tau_\beta \{\varphi_{\text{pre}}\} \tau \{\varphi_{\text{post}}\}$. First, without loss of generality, we assume that $\tau$ is in static single assignment (SSA) form; this ensures that variables are not assigned to more than once, simplifying our encoding. We also assume that $\varphi_{\text{pre}}$.
and $\varphi_{\text{post}}$ are logical formulas over program variables. Our encoding explicitly maintains failure probability using a special set of real-valued variables $\omega_i$, which encode failure probability after statement $s_i$ along $\tau$. In order to encode failure probability on unsatisfiable subtraces, we also use a special set of Boolean variables $h_i$ to track if an execution was blocked by an assume statement.

The function $\text{enc}$, defined in Fig. 9, is used to encode assignment, assume, and sampling statements; it maintains the variables $\omega_i, h_i$ and axiomatizes sampling statements using the aforementioned distribution axioms. Consider, for instance, the encoding for assignment statements: it constrains $\nu$ to $e$, while maintaining the same failure probability and blocked status, $\omega_i$ and $h_i$. Intuitively, the semantics of assignment statements is precisely captured by our logical encoding, so assignment statements do not increase the probability of failure. In contrast, the probability of failure increases when an axiom is applied for a sampling statement. Concretely, if the axiom family $\Pr_{\nu \sim d}[\varphi_{ax}] \leq e^{ub}$ is applied, we assume that $\neg \varphi_{ax}$ is true while accumulating probability of failure $e^{ub}$, as encoded in the constraint $\omega_i = \omega_{i-1} + e^{ub}$.

The following theorem formalizes the encoding of $\tau \vDash \varphi_{\text{pre}} \land \varphi_{\text{post}}$ and states its correctness.

**Theorem 6.4 (Soundness of Logical Encoding).** The specification $\tau \vDash \varphi_{\text{pre}} \land \varphi_{\text{post}}$ is valid if the following formula is satisfiable:

$$\forall V, \omega, h_1. \left( \varphi_{\text{pre}} \land \omega_0 = 0 \land h_0 = \text{false} \land \bigwedge_{i=1}^{n} \text{enc}(i, s_i) \right) \Rightarrow (\omega_n \leq \beta \land (\neg h_n \Rightarrow \varphi_{\text{post}})) \tag{2}$$

Observe that in the above encoding the only free symbols are the uninterpreted functions $f_1, \ldots, f_m$ introduced by the axiom families used in the encoding of sampling statements. Thus, checking satisfiability involves synthesizing interpretations for $f_1, \ldots, f_m$. (Equivalently, we can think of $f_1, \ldots, f_m$ as existentially quantified so that we check validity of $\exists f_1, \ldots, f_m \forall \nu \ldots$)

**Example 6.5.** Recall the trace $\tau_1$ from § 2.2 and Fig. 3 (program ex1), where we proved:

$$\tau_p \vDash \text{true} \xRightarrow{s_1} x \sim \text{bern}(0.5); \text{assume}(x); y \sim \text{bern}(p) \xRightarrow{s_2} \neg y$$

Using the encoding in Theorem 6.4, we get the following formula:

$$\forall x, y, p, \omega, h_1. \left( \omega_0 = 0 \land h_0 = \text{false} \land \bigwedge_{i=1}^{3} \text{enc}(i, s_i) \right) \Rightarrow (\omega_3 \leq p \land (\neg h_3 \Rightarrow \neg y))$$

To illustrate, $\text{enc}(1, x \sim \text{bern}(0.5))$ is the following constraint, using the axiom family in Table 2:

$$\omega_1 = \omega_0 + \begin{cases} 0.5 & \text{if } f_x(p) = 1 \\ 0.5 & \text{if } f_x(p) = 2 \\ 0 & \text{otherwise} \end{cases} + e^{ub} \land h_1 = h_0 \lor (f_x(p) = 1 \land x) \lor (f_x(p) = 2 \land \neg x)$$

The proof in § 2.2 used the interpretation $f_x(p) = 3$, allowing $x$ to take any value. 

6.2 From Synthesis to Craig Interpolation

Now that we have defined our logical constraints, we can apply Craig interpolation on the above encoding in Theorem 6.4 to construct the labeling functions, λ and κ, for an automaton accepting τ.

Craig Interpolants. The standard notion of sequence interpolants [McMillan 2006] generalizes Craig interpolants between two formulas to a sequence of unsatisfiable formulas in first-order logic.

Definition 6.6 (Sequence interpolants). Let \( \bigwedge_{i=1}^{n} \phi_i \) be unsatisfiable. There exists a sequence of formulas \( \psi_1, \ldots, \psi_n \) such that:

1. \( \phi_1 \Rightarrow \psi_1 \) and \( \psi_n \Rightarrow \text{false} \) are valid,
2. for all \( i \in (1, n) \), \( \psi_i \land \phi_{i+1} \Rightarrow \psi_{i+1} \) is valid, and
3. \( \text{fv}(\psi_i) \subseteq \text{fv}(\phi_1, \ldots, \phi_i) \cap \text{fv}(\phi_{i+1}, \ldots, \phi_n) \).

Note that sequence interpolation is equivalent to solving a form of recursion-free Horn clauses [Rümmer et al. 2013]; we use an interpolation-based presentation to reduce notational overhead.

Labeling Automata via Interpolation. Suppose that we have discovered interpretations for \( f_1, \ldots, f_m \) that satisfy Eq. (3) from Theorem 6.4. This implies that the following formula, which is Eq. (3) after negating it and instantiating \( f_1, \ldots, f_m \) with their interpretations, is unsatisfiable:

\[
\varphi_{\text{pre}} \land \omega_0 = 0 \land \bigwedge_{i=1}^{n} \text{enc}(i, s_i) \land \neg(\omega_n \leq \beta \land (\neg h_n \Rightarrow \varphi_{\text{post}}))
\]

It follows that we can construct a sequence of Craig interpolants for the following problem:

\[
\varphi_{\text{pre}} \land \omega_0 = 0 \land \text{enc}(1, s_1) \land \cdots \land \text{enc}(n, s_n) \land \neg(\omega_n \leq \beta \land (\neg h_n \Rightarrow \varphi_{\text{post}}))
\]

Every interpolant \( \psi_i \) encodes the set of reachable states and the failure probability after executing the first \( i \) program statements beginning from a state in \( \varphi_{\text{pre}} \). The free-variable condition for interpolants implies that the only free variables in \( \psi_i \) are \( h_i, \omega_i \), and live program variables after the \( i \)th statement. The challenge is that interpolants describe both the program state invariants and the failure probability invariants, corresponding to the \( \lambda \) and \( \kappa \) needed to label the failure automaton. Fortunately, these labels can be extracted from the interpolants. The following theorem formalizes the transformation and states its correctness.

Theorem 6.7 (Well-labelings from interpolants). Let \( \{\psi_i\}_i \) be the interpolants computed as shown above. Let \( A_\tau = (Q, \delta, \lambda, \kappa) \) be the failure automaton that accepts only the trace \( \tau = s_1, \ldots, s_n \), i.e., \( \delta = \{q^0 \xrightarrow{s_1} q_1, q_1 \xrightarrow{s_2} q_2, \ldots, q_{n-1} \xrightarrow{s_n} q^\text{ac}\} \). Set the labeling functions as follows:

1. \( \lambda(q^0) \triangleq \varphi_{\text{pre}} \) and \( \kappa(q^0) \triangleq 0 \).
2. \( \lambda(q_i) \triangleq \exists \omega_i, \psi_i[h_i \Rightarrow \text{false}] \) and \( \lambda(q^\text{ac}) \triangleq \exists \omega_n, \psi_n[h_n \Rightarrow \text{false}] \).
3. \( \kappa(q_i) \triangleq f(V_{\text{det}}) \), where \( f(V_{\text{det}}) \) is the function that returns, for any valuation of \( V_{\text{det}} \), the largest value of \( \omega_i \) that satisfies \( \exists \psi_i \). For \( \kappa(q^\text{ac}) \), we use \( \exists \psi_i \). Then, \( A_\tau \) is well-labeled and implies \( \varphi_{\text{pre}} \) \( \tau \varphi_{\text{post}} \).

Notice that for \( \lambda \) we set \( h_i \) to be \text{false}, since we are only interested in states that pass assume statements (reachable states). We existentially quantify \( \omega_i \), as it is not a program variable. Also notice the technicality in constructing \( \kappa \); this arises because the interpolant is a relation over values of \( \omega_i \) and \( V_{\text{det}} \), while the label of \( \kappa(q_i) \) is technically a function from \( V_{\text{det}} \) to [0,1]. In practice, we need not construct the function \( f \); we can perform all needed checks using relations.
We have implemented our approach atop the Z3 solver [de Moura and Bjørner 2008]. We encode statements using the following first-order theories: linear arithmetic, uninterpreted functions, and arrays. Below, we describe our implementation; we refer to the extended version [Smith et al. 2018] for further details.

Algorithmic Strategy. Our implementation is a determinization of the algorithm presented in §5. To ensure that we get tight upper bounds on failure probability, our implementation aggressively tries to apply the merge rule—recall that the merge rule allows us to take the maximum failure probability across two automata, instead of the sum. Specifically, we modify the rule trace to return a set of traces \(\tau_1, \ldots, \tau_n \in \mathcal{L}(P) \cap \mathcal{L}(A)\). Then, we attempt to simultaneously label all traces with the same interpolants at nodes pertaining to the same control location. To ensure that we compute similar interpolants across traces, we use the same distribution axiom for the same sampling instruction in all traces it appears in. Finally, we attempt to apply the rule generalizing to create cycles in the resulting automaton.

We elide \(\varphi'_T\) and \(\varphi'_L\), which are defined similarly.

Fig. 10. Privacy-preserving algorithms and their accuracy specifications

7 IMPLEMENTATION AND CASE STUDIES
7.1 Overview of Implementation
We have implemented our approach atop the Z3 SMT solver [de Moura and Bjørner 2008]. We encode statements using the following first-order theories: linear arithmetic, uninterpreted functions, and arrays. Below, we describe our implementation; we refer to the extended version [Smith et al. 2018] for further details.
Discovering Axioms. Given a formula of the form $\exists f. \forall X. \varphi$, we check its validity using a propose-and-check loop: (i) we propose an interpretation of $f$ and then (ii) check if $\forall X. \varphi$ is valid with that interpretation using the smt solver (more on this below). The first step proposes interpretations of $f$ of increasing size, e.g., for a unary function $f(x)$, it would try 0, 1, $x$, $x + 1$, etc. As we shall see, even for complex randomized algorithms from the literature, the required axioms are syntactically simple, so this simple strategy works rather well.

Checking Validity. The case studies to follow make heavy use of non-linear arithmetic (e.g., $\frac{x - y}{z} + u > 0$) and transcendental functions (namely, log). Non-linear theories are generally undecidable. To work around this fact, we implement an incomplete formula validity checker using an eager version of the theorem enumeration technique recently proposed by Srikanth et al. [2017]. First, we treat non-linear operations as uninterpreted functions, thus overapproximating their semantics. Second, we strengthen formulas by instantiating theorems about those non-linear operations. For instance, the following theorem relates division and multiplication: $\forall x, y. y > 0 \Rightarrow \frac{x \cdot y}{y} = x$. We then instantiate $x$ and $y$ with terms over variables in the formula. Since there are infinitely many possible instantiations of $x$ and $y$, we restrict instantiations to terms of size 1, i.e., variables/constants.

Our implementation uses a fixed set of theorems about multiplication, division, and logarithms. These are instantiated for every given formula, typically resulting in $\sim$1000 additional conjuncts.

Interpolation Technique. Given the richness of the theories we use, we found that existing proof-based interpolation techniques either do not support the theories we require (e.g., the MathSAT solver [Cimatti et al. 2013]) or fail to find generalizable interpolants, e.g., cannot discover quantified interpolants (e.g., Z3). As such, we implement a template-guided interpolation technique [Albarghouthi and McMillan 2013; Rummer and Subotic 2013], where we force interpolants to follow syntactic forms that appear in the program. Specifically, for every Boolean predicate $\varphi$ appearing in the program, the specification, or the axioms, we create a template $\varphi^t$ by replacing its variables with placeholders, denoted $\bullet_i$. For instance, given $x > y$, we generate the template $\bullet_1 > \bullet_2$.

Given a set of templates, our interpolation technique searches for an interpolant as a conjunction of instantiations of those templates, where each placeholder can be replaced by a well-typed term over formula variables. Given the infinite set of possible instantiations, our implementation fixes the size of possible instantiations (e.g., to size 1), and proceeds by finding the smallest possible interpolants in terms of number of conjuncts. If it cannot, it expands the search to terms of larger sizes.

7.2 Case Studies in Privacy-Preserving Algorithms

Differential privacy [Dwork et al. 2006] is a strong probabilistic property modeling statistical data privacy. Informally, a randomized database query satisfies differential privacy if adding/removing a single individual’s private data to/from the database does not change the output distribution very much, so that differentially private algorithms reveal little about any single individual’s record. To achieve this property, algorithms add random noise at key points in the computation. Sophisticated differentially private algorithms are known for a wide variety of common data analyses, and differential privacy is starting to see deployments in both industry [Erlingsson et al. 2014; Johnson et al. 2018] and government [Abowd and Schmutte 2017; Haney et al. 2017].

Intuitively, more random noise yields stronger privacy guarantees at the expense of accuracy—the noisy answers may be too far from the exact answers to be of any practical use. Therefore, the designer of a differentially private algorithm aims to maximize accuracy of the computed results while achieving some target level of privacy. We now consider a number of algorithms from the differential privacy literature and demonstrate our technique’s ability to automatically prove their
accuracy guarantees. The algorithms and their specifications are shown in Fig. 10 and described below; Table 3 provides runtime and other statistics, which we discuss later in this section.

**Randomized Response** (randResp). One of the oldest randomized schemes for protecting privacy is randomized response, proposed by Warner [1965] decades before the formulation of differential privacy. In the typical setting, an individual has a single bit (0 or 1) as their private data, representing e.g. the presence of some disease or genetic marker. Under randomized response, the individual flips two fair coins: if the first result is heads, the individual reports their bit honestly, otherwise they ignore their private bit and report the result of the second flip. In this way, randomized response guarantees a degree of privacy by introducing plausible deniability—an individual’s reported bit could have been the result of chance. At the same time, randomized response guarantees a weak notion of accuracy, as the output is biased towards the true private bit with probability 3/4. We encode randomized response as the first program in Fig. 10 and prove the accuracy guarantee. In the code, priv is the individual’s private bit. The program draws two bits uniformly and then decides what to return; fst and snd extract the results of the first and second bits, respectively. The accuracy guarantee states that the returned answer is equal to the true private bit, except with probability at most 1/4. Our implementation synthesizes the axiom $\text{priv} \iff \text{snd}(r)$; this ensures that the second bit has the same value as priv, so if the first bit is 0 and the else branch is taken, the algorithm is forced to return the right result, with a failure probability of 1/4.

**Noisy Sum** (noisySum). Our next algorithm computes the sum of a set of numeric queries, adding noise to the answer of each query in order to ensure differential privacy. This is a simplified version of the private counters by Chan et al. [2011] and Dwork et al. [2010], which are used to publish aggregate statistics privately, e.g., total number of website visitors.

The noisySum program takes three inputs: a set $Q$ of integer-valued queries, a database $d$ holding the private data, and a parameter $\varepsilon \in \mathbb{R}$ representing the desired level of privacy.\(^5\) The program populates an integer array $a$ with answers to each query, with noise drawn from the Laplace distribution with scale controlled by $\varepsilon$; smaller $\varepsilon$ is more private, but requires more noise. Finally, the output is the sum of all noisy answers.

The accuracy guarantee bounds how far the noisy sum deviates from the true sum with failure probability $p$, where $p$ is a parameter. Our implementation synthesizes an axiom for each Laplace sampling, setting the failure probability to be $p/|Q|$ each time. Therefore, at step $i$, $|a_i - Q_i(d)| \leq \frac{1}{\varepsilon} \log(|Q|/p)$. Since there are $|Q|$ iterations, after the loop exits we have $s - \sum_{j=1}^{\lfloor |Q| \rfloor} Q_j(d) | \leq \frac{|Q|}{\varepsilon} \log(|Q|/p)$ with a failure probability of at most $|Q| \cdot \frac{p}{|Q|} = p$.\(^5\)

**Report Noisy Max** (noisyMax) and the **Exponential Mechanism** (expMech). Our next pair of algorithms select an approximate maximum element from a set of private data.

\(^5\)We encode $Q$ as an integer array where index $i$ holds the result of $Q_i(d)$.
In Report Noisy Max [Dwork and Roth 2014], noisyMax in Fig. 10, the algorithm is presented with a set \( Q \) of integer queries, a private database \( d \), and a privacy level \( \varepsilon \). The algorithm then evaluates each query on \( d \) and adds Laplace random noise to protect privacy. Finally, the index of the query with the largest noisy value is returned. For example, if each query counts the number of patients with a certain disease, then Report Noisy Max will report a disease that may not be true most prevalent disease, but whose count is not too far from the true maximum count.

The postcondition states that the answer of the returned query \( Q_b \) is not too far below the answer of the actual maximum query. To achieve failure probability \( p \), our implementation synthesizes an axiom for the Laplace sampling statement with failure probability \( p/|Q| \). Since the loop executes \(|Q| \) times, we establish that the postcondition holds with probability \( p \). To do so, the interpolation engine discovers a number of key facts; we outline two of them:

\[
\forall j \in [1,i). |a_j - Q_j(d)| \leq \frac{2}{\varepsilon} \log \frac{|Q|}{p} \quad \text{and} \quad \forall j \in [1,i). a_b \geq a_j
\]

The first formula specifies that, for every element of \( j \) of \( a \), its distance from the corresponding valuation of \( Q_j(d) \) is bounded above by \( 2/\varepsilon \log |Q|/p \)—this follows directly from the choice of distribution axiom. The second formula states that the best element is indeed larger than all previously seen ones. Upon loop exit, these facts, along with others, are sufficient to imply the postcondition. Notice that the \( 2/\varepsilon \log |Q|/p \) in the first formula weakens to \( 4/\varepsilon \log |Q|/p \) in the postcondition. This is due to the two-sided error introduced by the absolute value in the Laplace axiom. The proof computed for noisyMax is presented in detail in the extended version [Smith et al. 2018].

The algorithm expMech is a discrete version of the seminal Exponential mechanism [McSherry and Talwar 2007], a fundamental algorithm in differential privacy. This algorithm is used to achieve differentially privacy in non-numerical queries, as well as a mechanism for achieving certain notions of fairness in decision-making algorithms [Dwork et al. 2012]. expMech takes a set \( R \) of possible output elements, a utility function \( u \) mapping each element of \( R \) and private database to a numeric score, a private database \( d \), and privacy parameter \( \varepsilon \). The algorithm aims to return an element of \( R \) that has large utility on the given database. expMech differs from noisyMax through the use of the exponential distribution; because the exponential distribution never produces results lower than the shift, the accuracy bound for the expMech is better. The distance to the true maximum is at most \( \frac{2}{\varepsilon} \log(2|R|/p) \) instead of \( \frac{4}{\varepsilon} \log(|Q|/p) \), with failure probability at most \( p \). To prove this, our implementation synthesizes an axiom analogous to that used for noisyMax.

**Above Threshold** (aboveT) and the **Sparse Vector Mechanism** (sparseVec). A useful differential privacy primitive is to return the first query in a list with a numeric answer (approximately) above some given threshold, ignoring queries with small answers. Our final two privacy examples do just this. The Above Threshold algorithm [Dwork and Roth 2014] takes a list \( Q \) of queries, a private database \( d \), a numeric threshold \( T \), and the target privacy level \( \varepsilon \). First, the program computes a noisy threshold \( t \) by adding noise to the true threshold \( T \). The program loops through the queries, comparing the noisy answer of each query to the noisy threshold. If the noisy answer is above the noisy threshold, the program sets the flag done and exits the loop. Finally, the algorithm returns the index of the approximately above threshold query, or a default value \( \perp \) if no such query was found.

The accuracy guarantee requires some care. There are two cases: the returned value is either a query index, or \( \perp \). In the first case, \( q_{\text{ans}} \) should have true value not too far below the exact threshold \( T \), and all prior queries should have true value not too far above \( T \). In the second case, no query was found to be above threshold after adding noise, so no true answer should be too far above \( T \). To prove this property, we synthesize axioms for the Laplace sampling instructions with...
different failure probabilities: $\frac{p}{2}$ for the threshold sampling, and $\frac{p}{2|Q|}$ for each loop sampling. There is one threshold sampling and at most $|Q|$ loop iterations, so the total failure probability is at most $\frac{p}{2} + |Q| \cdot \frac{p}{2|Q|} = p$.

A slightly more involved variant of this algorithm, called Numeric Sparse Vector [Dwork and Roth 2014], also returns a noisy answer to the above threshold query along with the query’s index. Again, the accuracy property describes the two cases—above threshold query found, and no above threshold queries. In both cases, the noisy query answer should be close to the true answer. The proof proceeds much like in the simpler variant, adjusting the failure probabilities when applying axioms in order to take the additional noisy answer sampling into account.

Discussion of Results. Table 3 summarizes the results of applying our implementation to the above algorithms. The table lists the synthesized axiom per sampling statement—recall that our implementation strategy forces different instances of a sampling statement to use the same axiom. Additionally, we list the number of proposed and checked axioms (PA), 6 the largest number of theorem instantiations for dealing with non-linear arithmetic (TI), and the total time in seconds.

Consider the aboveT algorithm. The implementation attempts 22 different pairs (because there are two sampling statements) of axioms. Table 3 lists the synthesized interpretation of the function $f(V^\text{in})$ for the first and second sampling statements. The implementation discovers the axiom that assigns a failure probability $p/2$ for the first sampling statement and $p/(2|Q|)$ for the second sampling statement. Proving accuracy of aboveT takes 23 seconds and 752 theorems are instantiated to interpret non-linear arithmetic. Notice that noisySum takes the longest amount of time, even though it only attempts 5 axioms. This is due to the large number (≈5500) of theorem instantiations. For sparseVec, the implementation proposes 941 axioms before discovering the shown axioms.

To the best of our knowledge, no existing tools can automatically reason about the algorithms and accuracy properties we have discussed here. The algorithms we considered are small yet sophisticated. As the number of sampling statements increases, the space of possible axioms grows combinatorially, impacting synthesis performance. As research into constraint-based program synthesis progresses, our approach can directly benefit from these developments.

7.3 Case Study in Unreliable Hardware

To demonstrate our approach’s versatility, we consider another possible application: analyzing programs executing on approximate hardware, which is unreliable but efficient.

We use the program searchRef from the Rely system by Carbin et al. [2013], shown in Fig. 11, which implements a pixel-block search algorithm from x264 video encoders. The program receives a constant number of pixel blocks ($nblocks = 20$) of size $16 \times 16$ ($height \times width$). This program is expected to provide adequate video encoding despite potential hardware failures. Rely’s programming model exposes unreliable arithmetic operations, denoted with a dot (e.g. $\dot{+}, \dot{-}$, etc.), which may fail with small probability (say, $10^{-7}$). Reading from variables typed as unreliable may also fail with a small probability. Rely assumes loops over unreliable data have a constant bound on the number of iterations, so these loops can be unrolled.

Our goal is to prove the probability of a reliable execution is at least 0.99, where reliability implies no failures along the execution. 7 To do so, we analyze a version of the program instrumented with a Boolean flag $unrel$, which is initialized to $false$. We model each unreliable operation by adding a sampling from the Bernoulli distribution to determine whether the operation fails. For instance, a read $y \leftarrow x$ from an unreliable $x$ is transformed into $y \leftarrow x; unrel \leftarrow unrel \lor \text{bern}(10^{-7})$. We then use our implementation to prove $\forall_{0.01} \{true\} \text{searchRef} \{unrel = false\}$.

---

6PA does not include the many possible axiom instantiations that are not well-typed.
7Rely multiplies this probability by the reliability of the inputs ($pblocks, cblock$)—this does not impact the analysis.
Unlike Rely, we do not assume independent failures. Our analysis thus gives a more conservative estimate of failure probability, but, as a benefit, retains soundness even if failures are correlated. Nevertheless, we are able to prove that the program is reliable with probability $> 0.992832$, compared to the $0.994885$ computed by Rely. Moreover, since our approach is symbolic, we can prove a symbolic reliability bound as a function of the number of blocks and their size. This allows us to ask: how many blocks can we use, and how large, and still be reliable? We automatically establish the parameterized failure probability $1.4 \cdot 10^{-6} \cdot \text{nblocks} \cdot \text{height} \cdot \text{width}$, describing how program parameters affect reliability. For instance, we can increase the number of blocks to 25 and still maintain $> 0.99$ reliability, or quadruple the size of each block to $32^2$ pixels and get $> .97$ reliability. In both settings, our approach completes the proof in less than 2 seconds.

8 RELATED WORK

Interpolation & Trace Abstraction. The idea of generalizing proofs of finitely many traces to a proof for all program traces has appeared in numerous guises in the software verification literature, e.g., [Albarghouthi et al. 2012; Ball and Rajamani 2001; Cook et al. 2005; Farzan et al. 2013; Heizmann et al. 2009, 2010, 2013; Henzinger et al. 2004; McMillan 2006]. We extend this idea to the probabilistic setting. Specifically, we (i) base our proof technique and algorithm on the trace abstraction framework [Heizmann et al. 2009, 2010, 2013], and (ii) employ interpolants, as used by McMillan [2006], to construct Hoare-like annotations of program traces.

The use of interpolants in software verification was first employed by Henzinger et al. [2004] for constructing predicate abstract domains in counterexample-guided abstraction refinement (CEGAR). McMillan’s work on lazy abstraction with interpolants [McMillan 2006] used proofs of correctness of program traces to directly construct Hoare-style annotations. In it, McMillan showed how to use the annotation of finitely many traces to construct an abstract reachability tree, which represents an annotated, finite unrolling of the program’s CFG.

The trace abstraction framework by Heizmann et al. [2009, 2010, 2013] provided an insightful and general view of interpolation-based verification through the lens of automata. As we have seen, trace abstraction extends naturally to handle probabilistic programs and properties: the

```
fun searchRef(pblocks, cblock)
  reliable: i, j, k
  unreliable: minssd, minblock, ssd, t, t1, t2
  i ← 0
  while i < nblocks do
    ssd, j ← 0, 0
    while j < height do
      k ← 0
      while k < width do
        t1, t2 ← pblocks[i, j], cblock[j]
        t ← t1 + t2
        ssd ← ssd + (t × t)
        k ← k + 1
        j ← j + 1
      if ssd < minssd then
        minssd, minblock ← ssd, i
      i ← i + 1
  return minblock
```

Fig. 11. Reliability example [Carbin et al. 2013]
automata-based view allows us to maintain sets of traces as labeled failure automata, constructed from the annotation of individual traces by careful application of our MERGE and GENERALIZE rules, and aggregate their probabilities of failure using the union bound.

**Deductive Probabilistic Verification.** Deductive verification techniques for probabilistic programs include probabilistic Hoare logics [Barthe et al. 2018; Chadha et al. 2007; den Hartog 2002; Rand and Zdancewic 2015]. and the lightweight probabilistic logic of Barthe et al. [2016b] our technique is closely related to, just as the classical interpolation-based techniques mirror Hoare-style proofs. Deductive techniques are highly expressive, but the complex proofs typically must be constructed manually or in an interactive setting. In contrast, our approach has the advantage of automation.

**Pre-Expectation Calculus.** The pre-expectation calculus and associated predicate transformers [Kozen 1985; Morgan et al. 1996] can prove properties of probabilistic programs, but have practical obstacles to full automation. Computing pre-expectations across sampling instructions yields an integral over the distribution being sampled from. Complex distributions, like the infinite-support Laplace distribution, yield correspondingly complex integrals that are difficult to reason about. Any automation of the pre-expectation calculus will need to establish algebraic properties about these mathematical expressions. Our use of distribution axioms (§ 6) obviates the need to reason directly about integrals via a reduction to synthesis.

**Martingales.** Martingales—probabilistic analogues of loop invariants—are used in automated tools to prove termination conditions [Chakarov and Sankaranarayanan 2013; Chaudhary et al. 2016a,b, 2017; McIver et al. 2018] and properties of expected values [Barthe et al. 2016a; Chakarov and Sankaranarayanan 2013]. Automated martingale synthesis techniques are restricted to linear or polynomial invariants, which alone are unable to prove the accuracy properties we are interested in. Further, in such techniques, even after an invariant is inferred, additional manual probabilistic reasoning is needed to establish a postcondition—for example, Chakarov and Sankaranarayanan [2013] apply the Azuma–Hoeffding inequality to the inferred expectation invariant.

**Probabilistic Model Checking.** Probabilistic model checking is perhaps the most well-developed technique for automated reasoning about probabilistic systems. Traditionally, it focused on temporal properties of Markov Decision Processes (MDP)—surveys by Kwiatkowska et al. [2010] and Katoen [2016] overview the current state of the art.

Our program model can be cast as an infinite-state MDP, with non-determinism at program entry to pick an initial state. There have been a number of abstraction-based techniques for reducing the size of large (or infinite) MDPs [Hermans et al. 2008; Kattenbelt et al. 2009, 2010]. To our knowledge, existing works cannot handle the programs and properties we consider here. The general limitation is the inability of existing model checking techniques to handle distribution expressions—e.g., a Laplace whose scale is a parameter—and failure probabilities that are expressions. Probabilistic CEGAR [Hermans et al. 2008] uses a guarded-command language where probabilistic choice is a real-value determining the probability of executing each command. Other techniques limit distribution expressions to finite distributions with constant parameters [Kattenbelt et al. 2009].

Teige and Fränzle [2011] consider interpolation in stochastic Boolean satisfiability [Littman et al. 2001], where formulas contain existential and probabilistic quantifiers. The approach has been used for generalizing bounded encodings of finite-state MDPs, in an analogous fashion to the original work on interpolation-based model checking [McMillan 2003].

**Other Probabilistic Analyses.** Probabilistic abstract interpretation [Cousot and Monerau 2012] generalizes the abstract interpretation framework to a probabilistic setting; other techniques can be
cast in this framework [Claret et al. 2013; Monniaux 2000, 2001, 2005]. Recently, Wang et al. [2018] presented pMAF, an elegant algebraic framework for constructing analyses of probabilistic programs. The approach is rather general, accepting recursive programs and supporting interprocedural analyses. Unlike pMAF, whose results depend highly on the expressiveness of the chosen abstract domain, our technique constructs abstractions on demand, à la interpolation-based verification, at the risk of never generalizing. pMAF instantiations considered by Wang et al. [2018] cannot prove our target accuracy properties, but alternative instantiations might achieve something similar.

Another line of work reduces probabilistic verification to a form of counting [Albarghouthi 2017; Belle et al. 2015; Chistikov et al. 2015; Mardziel et al. 2011]. To compute the probability that a formula is SAT, these techniques count the number of satisfying assignments—or perform numerical volume estimation in the infinite-state case. While these techniques can compute very precise—often exact—probabilities, they target simpler program models. Specifically, programs have no inputs, probability distribution are not parameterized, and loops are handled via unrolling.

Our technique is related to works verifying relational probabilistic properties, including differential privacy and uniformity [Albarghouthi and Hsu 2018a,b]. These systems encode the space of coupling proofs as a constraint-based synthesis problem. Our technique handles different properties, but shares the high-level design principle of reducing probabilistic reasoning to logical reasoning.

Computer algebra and symbolic inference methods (e.g., [Cusumano-Towner et al. 2018; Gehr et al. 2016; Narayanan et al. 2016]) have been applied to probabilistic programs in different domains (e.g., [Gehr et al. 2018]). While these tools can automatically generate symbolic representations of output distributions, proving properties about these distributions remains challenging. Modern implementations use a variety of custom heuristics and reduction strategies to try to simplify complex algebraic terms, a computationally-expensive task.

9 CONCLUSIONS AND FUTURE DIRECTIONS

We have presented a generalization of trace abstraction for proving accuracy properties of probabilistic programs. This required four key ideas: (i) representing probabilistic traces with failure automata labeled with formulas and probabilities, (ii) merging and generalizing these failure automata, (iii) axiomatizing distributions by solving a synthesis problem, and (iv) applying Craig interpolation to construct labels of failure automata. These ideas enable automated verification of accuracy properties using logic-based techniques, while handling rich programs and properties.

For future work, we see trace abstraction modulo probability being extended with other kinds of probabilistic reasoning, perhaps based on independence or expectations; the challenge is keeping the complexity of the logical encoding under control. Another natural path is to connect to recent work on probabilistic abstract interpretation, e.g. by Wang et al. [2018]. One could imagine enhancing a standard abstract domain with failure probabilities. However, the treatment of sampling instructions is unclear—there are typically multiple axioms for a given distribution, and a proof may need several different axiomatizations to prove a target property.

ACKNOWLEDGMENTS

We thank Thomas Reps, Zachary Kincaid, Andreas Podelski (our shepherd), and the anonymous referees for their insightful comments on earlier drafts of this work. This work is supported by the National Science Foundation CCF under Grant Nos. 1566015, 1704117, 1652140, and 1637532.

REFERENCES