# Synthesis of Probabilistic Privacy Enforcement

Martin Kučera, Petar Tsankov, Timon Gehr, Marco Guarnieri, Martin Vechev
ETH Zurich
firstname.lastname@inf.ethz.ch

## ABSTRACT

Existing probabilistic privacy enforcement approaches permit the execution of a program that processes sensitive data only if the information it leaks is within the bounds specified by a given policy. Thus, to extract any information, users must manually design a program that satisfies the policy.

In this work, we present a novel synthesis approach that automatically transforms a program into one that complies with a given policy. Our approach consists of two ingredients. First, we phrase the problem of determining the amount of leaked information as Bayesian inference, which enables us to leverage existing probabilistic programming engines. Second, we present two synthesis procedures that add uncertainty to the program's outputs as a way of reducing the amount of leaked information: an optimal one based on SMT solving and a greedy one with quadratic running time.

We implemented and evaluated our approach on 10 representative programs from multiple application domains. We show that our system can successfully synthesize a permissive enforcement mechanism for all examples.

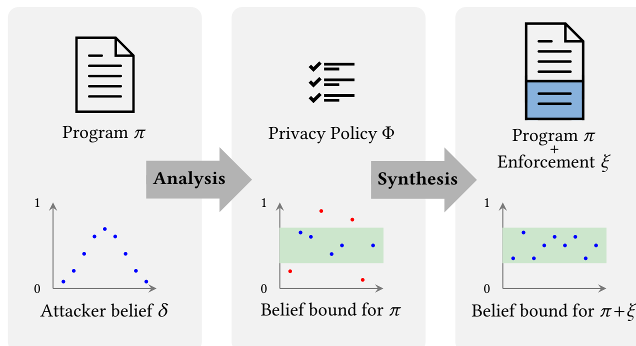



**Figure 1: The two ingredients of our approach: *probabilistic analysis* to determine the information leaked by the program about the secret, and *enforcement synthesis* that bounds the leakage according to the policy bounds.**

## 1 INTRODUCTION

Privacy enforcement systems, i.e. systems that protect the privacy of sensitive data with respect to policies, must be both *permissive* and *secure*. That is, users should be permitted to process sensitive data while making sure they cannot learn too much information about the sensitive data, thereby violating privacy policies. In the context of genomic privacy, for instance, it is important to allow medical researchers to process and aggregate genomic data as this can be extremely valuable for their work; however, it is also critical to preserve the privacy of the patients [3, 30]. This tension between permissiveness and security of privacy enforcement permeates many practical domains, including medical data protection [31], location privacy in location-based services [29, 35, 59], and privacy of personal data in social networks [39].

***Existing Approaches.*** Existing access-control solutions [9, 53, 54] can only enforce basic all-or-nothing policies for a particular piece of sensitive data (hereafter called *the secret*). They are thus often overly-restrictive in practice; see [37].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*CCS '17, October 30-November 3, 2017, Dallas, TX, USA*
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3134079

Recently, probabilistic privacy enforcement (PPE) approaches (e.g. [37]) have been proposed as a promising step towards improving the permissiveness of privacy enforcement. PPE approaches can enforce a wide range of privacy policies that *bound* how much an attacker can *learn* about the secret. For example, *"no medical researcher can correctly guess that Alice has disease X with probability higher than* 0.9*"*. To enforce such policies, PPE systems explicitly model the attacker's belief as a distribution over the possible values the secret can take. The attacker asks the PPE system to run a program (i.e., submits a query) that takes the secret as input. The PPE system then reveals the program's output to the attacker only if this does not leak too much about the secret; otherwise, the PPE system rejects the attacker's program.

Existing PPE systems reject a program if the program would leak information for some possible secret. As an example, suppose a medical researcher asks for the number of patients who have a particular disease. Further, suppose the PPE system must enforce that researchers cannot correctly guess that Alice (who is among the patients) has the disease with probability higher than 0.9. If the program outputs the total number of patients, then the researcher could potentially learn that all patients (including Alice), have the disease. The PPE system thus rejects this program.

***Problem Statement.*** Since existing PPE systems would reject any program that leaks too much information, a user must manually modify the program, e.g. by perturbing its output [19, 48]. Designing a policy-compliant program is hard and requires nontrivial probabilistic reasoning. This puts a significant burden on users who would like to process sensitive data.

In this work, we explore the problem of automatically transforming programs into policy-compliant ones. More formally, we address the following synthesis problem: *Given a program $\pi$, an*

*attacker belief $\delta$, and a privacy policy $\Phi$, transform $\pi$ into a program $\pi'$ that is guaranteed to satisfy the privacy policy $\Phi$ for the given attacker belief $\delta$.*

***This Work.*** In this paper, we propose the first solution to the problem of transforming a program into a policy-compliant one, where the policy is defined as a set of probabilistic assertions on the distribution over the program inputs (capturing the attacker belief). Our approach consists of two key ingredients, depicted in Figure 1. First, we phrase the problem of determining how much information the program's outputs leak about the secret as *probabilistic analysis*, and check whether the leakage is within the bounds specified by the policy; we depict the safe bounds in green in Figure 1. Second, to enforce the policy and reduce the amount of leaked information, the key idea is to *synthesize an enforcement* that transforms the program by adding uncertainty to its outputs. We show that solving this problem optimally is NP-equivalent and present an algorithm using a reduction to linear optimization over SMT constraints.

***Main Contributions.*** Our main contributions are:

- A formulation of the permissive privacy enforcement synthesis problem (Section 4).
- An optimal synthesis algorithm based on a reduction to linear optimization over SMT constraints (Section 5).
- A quadratic-time greedy synthesis algorithm that is sound but not guaranteed to be optimal (Section 6).
- An end-to-end implementation of our approach in a system called Spire[1] (Section 7).
- An evaluation of Spire on 10 representative programs from multiple application domains. We show that our system can successfully synthesize a permissive enforcement mechanism for all examples (Section 8).

# 2 OVERVIEW

In this section, we first present a simple, but illustrative, example. We then describe the probabilistic privacy model, which we borrow from [37], and we illustrate our enforcement synthesis approach. Finally, we present our attacker model.

## 2.1 Genomic Privacy Example

Genomic data is extremely valuable to medical researchers. Unfortunately, it also reveals sensitive personal information, such as predisposition to various diseases [2].

In this example, we consider the position rs11200638 in the HTRA1 gene [17]. At this position, each person has one of the following combinations of nucleotides: *AA*, *AG*, or *GG*, where *A* stands for adenine and *G* for guanine. A person who has the combination *AA* is 10 times more likely to develop Age-Related Macular Degeneration (ARMD), a medical condition that may result in blurred vision or blindness [41].

Patients can usually choose whether they want nucleotides at sensitive positions to remain private while their data is processed. Protecting the privacy of genomic data is, however, extremely challenging. A patient's genomic data is correlated with that of the patient's relatives, which enables highly nontrivial probabilistic inference attacks [3, 30, 31].

[1] Available at: http://www.srl.inf.ethz.ch/probabilistic-security

In our example, we consider three patients—Alice, Bob, and their child Carol—who have identified that their nucleotides at position rs11200638 are *GG*, *AA*, and *AG*, respectively. Carol's nucleotides are inherited by randomly selecting one from Alice and one from Bob. The nucleotides of Alice, Bob, and Carol are therefore statistically correlated. For instance, anyone who knows that the nucleotides of Alice and Carol are *GG* and *AG*, respectively, can infer that Bob must have at least one adenine nucleotide. Bob wants to ensure that no one can correctly guess that his nucleotides are *AA* with a probability higher than 0.75. Since Alice and Carol's genomic data reveal information about Bob, it is insufficient to protect Bob's data alone to enforce his policy.

## 2.2 Probabilistic Privacy Model

We adopt the probabilistic privacy model of [37], which can capture numerous practical scenarios. We informally describe the components of the model on our motivating example. We formally define this model in Section 3.

***Secret and Attacker Belief.*** The secret is a (sensitive) value that must be protected from the attacker. The attacker belief about the secret is then modeled as a probability distribution over all possible values the secret can take. We remark that in many settings it is realistic to precisely model the attacker belief; for example, whenever the secret is drawn from a well-known distribution, such as census data, genomic data, and so forth.

In our example, the secret consists of Alice, Bob, and Carol's nucleotides, and the attacker belief assigns a probability to each possible assignment of nucleotides for Alice, Bob, and Carol. We capture this distribution as a probabilistic program, given by the function `belief()` in Figure 2(b). In the program, we encode the nucleotides of Alice, Bob, and Carol with a two-dimensional array `nucl` that consists of three pairs of random variables (one pair per patient). Each of these random variables takes the value 0 or 1, which we interpret as the adenine and guanine nucleotides, respectively.

The random variables that capture Alice and Bob's nucleotides are initialized as Bernoulli random variables that take the value 1 with probability 0.77 and the value 0 with probability 0.23. The value 0.77 captures the frequency of guanine at position rs11200638, as reported in [1]. Lines 7-8 specify that Carol inherits her nucleotides from Alice and Bob randomly.

According to the attacker belief, the probability that Bob's nucleotides are *AA*, which corresponds to the probability of the event `nucl[Bob] = [A,A]`, is 0.0529.

***Program.*** The attacker asks the system to run a program (e.g. a query) that takes the secret as input. Suppose the attacker asks to run the program that returns the number of adenine nucleotides found at the rs11200638 positions in the HTRA1 genes of Alice, Bob, and Carol; see Figure 2(a). For our example, this program returns the value 3 because the secret is `nucl=[[A,A],[G,G],[A,G]]`.

Based on the observed output, the attacker revises her belief about the secret using Bayesian inference. The posterior distribution can be computed using state-of-the-art probabilistic solvers, such as [22, 40, 47]. For example, we can compute that, according to the attacker's revised belief, the probability that Bob's nucleotides are *AA* is 0.25, which is higher compared to her prior belief of 0.0529.

```
1   // Secret: nucl = [[A,A], [G,G], [A,G]]
2   def main(nucl: R[][]) {
3     A := 0; G := 1;
4     sum := 0;
5     for patient in [0..3) {
6       for position in [0..2) {
7         if (nucl[patient][position] == A) {
8           sum += 1;
9     } } }
10    // always outputs the exact sum
11    return sum;
12  }
```

**(a) (Original) program**

```
1   // Returns nucl[<Patient>][<Pos>] = <Nucl>
2   def belief() {
3     Alice := 0; Bob := 1; Carol := 2;
4     nucl := array[3][2];
5     nucl[Alice] = [flip(0.77), flip(0.77)];
6     nucl[Bob] = [flip(0.77), flip(0.77)];
7     C0 := nucl[Alice][flip(0.5)];
8     C1 := nucl[Bob][flip(0.5)];
9     nucl[Carol] = [C0, C1];
10    return nucl;
11  }
```

**(b) Attacker belief**

```
1   (nucl[Bob] == [A,A], [0,0.75])
```

**(c) Privacy policy**

**Figure 2: Example program, attacker belief, and privacy policy, written in the Psi language, presented in Appendix B.**

***Privacy Policy.*** The privacy policy can be captured by a set of probabilistic assertions over the attacker's belief. We capture such policies as predicates over the possible values of the secret together with lower and upper-bounds on the probabilities of these predicates; see Figure 2(c).

***Privacy Policy Enforcement.*** Existing enforcement systems, such as [37], run the attacker's program if for all possible outputs the privacy policy is satisfied according to the attacker's revised belief and otherwise they reject the program. These approaches reject the program because for the output 6 the probability that Bob's nucleotides are *AA* is 1, which is above the bound of 0.75. Note that the existing enforcements reject the program although the program would return 3 since the secret is `nucl = [[GG],[AA],[AG]]`, and this output does not result in a policy violation.

## 2.3 Synthesis of Permissive Privacy Enforcement

We propose a novel synthesis approach for enforcing privacy policies in a more *permissive* way. The key idea is to *synthesize an enforcement* for the given program which guarantees that the privacy policy is satisfied for the given attacker belief. To this end, the enforcement modifies the program by conflating certain outputs (e.g. outputs that result in policy violations) and makes them equally likely. We remark that this is a common approach to add uncertainty to the program output in order to leak less information [37, 50], as we discuss in Section 4. Below, we informally illustrate this idea on our motivating example. We formalize our notion of enforcement in Section 4.3 and prove its completeness in Section 4.4.2.

```
1   def main(nucl: R[][]) {
2     A := 0; G := 1;
3     sum := 0;
4     for patient in [0..3) {
5       for position in [0..2) {
6         if (nucl[patient][position] == A) {
7           sum += 1;
8     } } }
9     if (sum == 5 || sum == 6) {
10      // outputs that the sum is either 5 or 6
11      return pick([5, 6]);
12    } else {
13      // outputs the exact sum
14      return sum;
15    }
16  }
```

**Modified program**

**Figure 3: Modified program that satisfies the privacy policy for the given attacker belief (see Figure 2). The unchanged code is grayed out to highlight the synthesized enforcement (Lines 9-15). The enforcement conflates outputs 5 and 6.**

***Synthesized Enforcement.*** In our example, Bob's privacy policy is violated if the program given in Figure 2(a) returns the output 6. To avoid this behavior, our Synthesis of PermissIve pRivacy Enforcement (Spire) system synthesizes an enforcement that conflates the outputs 5 and 6 and makes them equally likely for each input. In Figure 3, we show the resulting program which is obtained by modifying the `return` statement of the original program (given in Figure 2(a)). Lines 9-15 in the new program illustrate the synthesized enforcement. Note that whenever the sum of adenine nucleotides is 5 or 6, the new program's output is 5 with probability 0.5 and 6 with probability 0.5. This behavior is implemented using the expression `pick([5, 6])` at Line 11. For all remaining sums of adenine the new program outputs the exact number.

The new modified program satisfies Bob's privacy policy for all outputs. In particular, for the output 6, the probability of the event `nucl[Bob] == [A,A]` is 0.56, which is below the bound of 0.75 defined in the privacy policy. We remark that the synthesis step is independent of the secret and therefore does not leak any additional information about the secret.

***Challenge.*** A trivial solution to enforcing the policy is to conflate all outputs. All outputs then leak no information about the secret. A key challenge is thus to synthesize an *optimal* enforcement, i.e. one that conflates as few outputs as possible. In our example, outputs $0, \dots, 4$ are not merged together with other outputs. We formalize two notions of optimality, called permissiveness and answer-precision, in Section 4.2 and discuss the complexity of finding an optimally permissive enforcement with respect to these notions in Section 4.4. We reduce the synthesis problem to a linear optimization over SMT constraints in Section 5 and give an efficient greedy heuristic that runs in quadratic time in Section 6.

***Attacker Model.*** We consider an attacker who: *(i)* can select any program (deterministic or probabilistic) that takes the secret as

input, and can ask the system to run it, *(ii)* can observe the output returned by the system, *(iii)* knows the privacy policy and the synthesis algorithm used to enforce it. Our synthesis algorithms are deterministic, and so assumption *(iii)* implies that the attacker knows that the system provides an output produced by running the program in Figure 3.

We work with probabilistic programs, as queries often add random noise to the output; e.g. see the examples in [48]. Furthermore, the computation in many privacy-relevant settings is probabilistic. Deterministic programs are a special case.

We assume that the attacker belief is known. This distribution captures common knowledge about the secret and is usually publicly available. For instance, the frequency of nucleotides in human genes can be found in public databases [1]. We remark that having an attacker belief (or a set of beliefs) is necessary when it comes to enforcing privacy policies formulated as bounds on the attacker belief [37]. This is because no enforcement can satisfy a non-trivial policy (i.e. a policy that is not satisfied by all programs) for all possible attacker beliefs.

The attacker can iteratively ask the system to run multiple programs against the secret. Enforcing the privacy policy in this setting requires tracking the attacker belief as outputs are revealed. We describe how the Spire system handles this iterative setting in Section 7, and in Section 8.2.4 we present experiments to evaluate the decrease in permissiveness of the synthesized enforcement after each iteration.

***Security Applications.*** Our synthesis approach can be used to restrict how much attackers can learn about sensitive data in numerous practical scenarios. It can be used, for instance, to enforce security properties such as opacity [48], k-anonymity [46, 51], and knowledge-based security [37]. Note that these are general security properties that are employed to enforce privacy in numerous relevant application domains. Examples include privacy in anonymous communication networks [50], genomic data privacy [2, 30], and privacy of location-based services [34, 39, 52]. In Section 8, we evaluate our implementation on examples related to genomic data, location data, and personal information, that we adopted from the literature. We discuss related work in Section 9.

## 3 PROBABILISTIC PRIVACY MODEL

In this section, we first introduce our notation and then present the probabilistic privacy model.

***Notation.*** Given two sets $\mathcal{I}$ and $\mathcal{O}$, we write $\mathbb{M}_{\mathcal{I}\times\mathcal{O}}$ to denote the set of all matrices over $\mathbb{R}$ whose rows and columns are indexed by $\mathcal{I}$ and $\mathcal{O}$, respectively. For a matrix $m \in \mathbb{M}_{\mathcal{I}\times\mathcal{O}}$, we write $m(o \mid i)$, where $i \in \mathcal{I}$ and $o \in \mathcal{O}$, to denote the element in row $i$ and column $o$.

We denote the set of all probability distributions over a set $S$ by $\mathcal{D}(S)$. A random variable $X : \Omega \to \mathcal{X}$ is a measurable function associating to each outcome $\omega \in \Omega$ a value $X(\omega) \in \mathcal{X}$, where $\mathcal{X}$ is a measurable space. Given a random variable $X : \Omega \to \mathcal{X}$ and a value $x \in \mathcal{X}$, we denote the event $\{\omega \in \Omega \mid X(\omega) = x\}$ by $X = x$.

Given an equivalence relation $\xi \subseteq S \times S$ over a set $S$, we write $[s]_\xi = \{s' \in S \mid (s, s') \in \xi\}$ for the equivalence class of an element $s \in S$ according to $\xi$, and we denote the quotient set of $\xi$ by $S/\xi = \{[s]_\xi \mid s \in S\}$.

***Probabilistic Programs.*** Let $\mathcal{I}$ and $\mathcal{O}$ be finite input and output sets, respectively. Following [13, 49], we define a *probabilistic program* as a stochastic matrix $\pi \in \mathbb{M}_{\mathcal{I}\times\mathcal{O}}$ where for each input $i \in \mathcal{I}$, we have $\sum_{o\in\mathcal{O}} \pi(o \mid i) = 1$. The element $\pi(o \mid i)$ denotes the probability that the program outputs $o \in \mathcal{O}$ given the input $i \in \mathcal{I}$. That is, for each input $i \in \mathcal{I}$, the program defines a distribution over the outputs $\mathcal{O}$.

***Secret and Attacker Belief.*** A *secret* is a value $i \in \mathcal{I}$ from the set of inputs. For example, if $\mathcal{I}$ is the set of all possible nucleotide sequences in a given gene, then the secret is a particular nucleotide sequence.

An *attacker belief* $\delta \in \mathcal{D}(\mathcal{I})$ is a distribution over the inputs $\mathcal{I}$ [14, 37]. The attacker belief captures the attacker's view on the likelihood that a particular value $i \in \mathcal{I}$ is the secret; i.e., the attacker belief of the secret $i \in \mathcal{I}$ is $\delta(i)$.

***Bayesian Inference.*** The attacker can ask the system to run a program $\pi$ that takes the secret as input. She observes the program's output and *revises* her belief about the secret based on the observed output, as described in Section 2.3. We rely on Bayesian conditioning to revise an attacker belief given the observed output.

We now capture the above notions with a probability space. Given a probabilistic program $\pi \in \mathcal{M}_{\mathcal{I}\times\mathcal{O}}$ and an attacker belief $\delta \in \mathcal{D}(\mathcal{I})$, we construct the probability space $(\Omega, \mathcal{F}, \mathbb{P}^\pi_\delta)$ with a sample space $\Omega = \mathcal{I} \times \mathcal{O}$, a set of events $\mathcal{F} = \mathcal{P}(\Omega)$, and a probability measure $\mathbb{P}^\pi_\delta(i, o) = \delta(i) \cdot \pi(o \mid i)$. In our overview example, the set of inputs is $\mathcal{I} = \{\mathtt{A}, \mathtt{G}\}^6$, where each six-tuple identifies the pairs of nucleotides of Alice, Bob, and Carol, and the set of outputs is $\mathcal{O} = \{0, \ldots, 6\}$. We represent the program's inputs and outputs with the random variables $I : \Omega \to \mathcal{I}$ and $O : \Omega \to \mathcal{O}$, respectively, where $I(i, o) = i$ and $O(i, o) = o$. Note that for a probabilistic program $\pi$, we have $\mathbb{P}^\pi_\delta(O = o \mid I = i) = \pi(o \mid i)$. Furthermore, for an attacker belief $\delta$, we have $\mathbb{P}^\pi_\delta(I = i) = \delta(i)$.

After observing an output $o$, the attacker *revises* her belief $\delta$ to $\delta'$, where the revised belief $\delta'$ is given by the distribution $\delta'(i \mid o) = \mathbb{P}^\pi_\delta(I = i \mid O = o)$. This distribution is computed using the Bayes rule as follows:

$$\mathbb{P}^\pi_\delta(I = i \mid O = o) = \frac{\mathbb{P}^\pi_\delta(O = o \mid I = i) \cdot \mathbb{P}^\pi_\delta(I = i)}{\mathbb{P}^\pi_\delta(O = o)} = \frac{\pi(o \mid i) \cdot \delta(i)}{\sum_{i\in\mathcal{I}} \pi(o \mid i) \cdot \delta(i)}$$

***Privacy Policies.*** A *belief bound* is a pair $\varphi = (S, [a, b])$ where $S \subseteq \mathcal{I}$ is a subset of inputs $\mathcal{I}$ and $[a, b]$ is an interval such that $a, b \in \mathbb{Q}$ and $0 \le a \le b \le 1$ [37]. Given a program $\pi$, an attacker belief $\delta$, and a belief bound $\varphi = (S, [a, b])$, we say that $\pi$ *satisfies* $\varphi$ *for* $\delta$, denoted by $\pi, \delta \models \varphi$, if for all outputs $o \in \mathcal{O}$, we have

$$\mathbb{P}^\pi_\delta(I \in S \mid O = o) \in [a, b]$$

That is, for any output $o \in \mathcal{O}$ the program may return, the revised attacker belief (after observing $o$) about the predicate $S$ must be within the bounds $a$ and $b$. We remark that the program *must* satisfy the security assertion for all program outputs in order to allow an attacker to run the program. For further details, see [37]. Note that the definition generalizes the notion of opacity to the probabilistic

| | Outputs $O$ | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| AA AA AA | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| AA AA AG | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| GG GG GG | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

(a) Program $\pi$

| | Outputs $O/\xi$ | | | | | |
|---|---|---|---|---|---|---|
| | {0} | {1} | {2} | {3} | {4} | {5, 6} |
| AA AA AA | 0 | 0 | 0 | 0 | 0 | 1 |
| AA AA AG | 0 | 0 | 0 | 0 | 0 | 1 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| GG GG GG | 1 | 0 | 0 | 0 | 0 | 0 |

(b) Program $\textsc{Enf}(\pi, \xi)$

| | Outputs $O$ | | | | | | |
|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| AA AA AA | 0 | 0 | 0 | 0 | 0 | 0.5 | 0.5 |
| AA AA AG | 0 | 0 | 0 | 0 | 0 | 0.5 | 0.5 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| GG GG GG | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

(c) Program $\textsc{EnfRnd}(\pi, \xi)$

**Figure 4: The two kinds of enforcement Enf and EnfRnd illustrated on the motivating example program $\pi$ and the enforcement $\xi$ with equivalence classes $O/\xi = \{\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5, 6\}\}$. The encodings of $\pi$ and $\textsc{EnfRnd}(\pi, \xi)$ are in Figure 2(a) and Figure 3, respectively. We highlight outputs that violate the policy in red and outputs that are fused together in green.**

case and bounds the probability that the attacker can correctly guess the secret [49].

A *privacy policy* is a set $\Phi = \{\varphi_1, \ldots, \varphi_n\}$ of belief bounds. A program $\pi$ satisfies a privacy policy $\Phi$ for a given attacker belief $\delta$, denoted by $\pi, \delta \models \Phi$, if we have $\pi, \delta \models \varphi$ for every $\varphi \in \Phi$.

## 4 PERMISSIVE PRIVACY ENFORCEMENT

In this section, we present our notion of enforcement. We consider a common type of privacy enforcement where outputs that leak too much information are replaced by more ambiguous answers (e.g. by a set of outputs). The idea to enforce a policy by conflating outputs generalizes many existing privacy enforcement mechanisms from the literature. In [37], for example, programs with binary output are secured by conflating the outputs "Yes" and "No" into a "Deny answer" output whenever one of the outputs violates the policy. In the area of anonymous communication networks [50], researchers have explored the idea of conflating real traffic with fake traffic to leak less information to the attacker, hence effectively enforcing the policy. Finally, in database privacy, $k$-anonymity [46] is a key concept guaranteeing that any record of a released anonymized data set can not be tied to any less than at least $k$ records in the original private data set. A common technique for achieving $k$-anonymity is generalization [45, 51]: instead of releasing an exact value of an attribute (say, age of a patient is 47 years old), the value is generalized to a range (e.g. age is said to be in the range 40–49).

In the following, we first define what we mean by optimal enforcement. Afterwards, we formulate our enforcement synthesis problem and discuss its complexity.

### 4.1 Enforcement

If a program $\pi$ violates a privacy policy $\Phi$ for an attacker belief $\delta$, then we cannot permit the attacker to observe $\pi$'s output because, for some outputs, the revised attacker belief violates one or more belief bounds in $\Phi$. That is, there is an output $o \in O$ and a belief bound $(S, [a, b]) \in \Phi$ for which $\mathbb{P}^{\pi}_{\delta}(I \in S \mid O = o) \notin [a, b]$.

To satisfy the policy $\Phi$ for the given attacker belief $\delta$, we need to modify the program $\pi$. We introduce the notion of enforcement to delimit the space of possible modifications that can be applied to the program $\pi$. An enforcement modifies the program $\pi$ by conflating certain outputs and making them equally likely. In the overview example, the enforcement conflates the outputs 5 and 6: if the original program (given in Figure 2(a)) returns 5 or 6 with different probabilities for a given input, then the modified program (given in Figure 3) returns 5 with probability 0.5 and 6 also with probability 0.5 for the given input. This notion of enforcement can be implemented syntactically by modifying the `return` statement(s) of the original program (Line 11 in Figure 2(a)). At the semantic level, the sets of outputs fused together by the enforcement can be formalized as an equivalence relation over the outputs.

Let $\pi \in \mathcal{M}_{\mathcal{I} \times O}$ be a probabilistic program. An *enforcement for* $\pi$ is an equivalence relation $\xi \subseteq O \times O$. A program $\pi$ together with an enforcement $\xi$ gives us the program $\pi' \in \mathbb{M}_{\mathcal{I} \times (O/\xi)}$ that returns equivalence classes from $O/\xi$. The program $\pi' \in \mathbb{M}_{\mathcal{I} \times (O/\xi)}$, denoted by $\textsc{Enf}(\pi, \xi)$, is defined as follows:

$$\textsc{Enf}(\pi, \xi)(E \mid i) = \sum_{o \in E} \pi(o \mid i)$$

That is, given an input $i \in \mathcal{I}$, the probability that the program $\textsc{Enf}(\pi, \xi)$ outputs the equivalence class $E$ is the sum of the probabilities that $\pi$ returns an output $o \in E$.

To illustrate, in Figure 4(a) we depict the semantics of the program $\pi$ given in Figure 2(a), and in Figures 4(b) the semantics of the program $\textsc{Enf}(\pi, \xi)$.

The program $\textsc{Enf}(\pi, \xi)$ has a different signature than $\pi$, since the outputs of $\textsc{Enf}(\pi, \xi)$ are the set of equivalence classes $O/\xi$, while the outputs of $\pi$ are $O$. We can, however, also enforce $\xi$ without changing the signature of $\pi$. For instance, if for an input $i$ the program $\textsc{Enf}(\pi, \xi)$ outputs the equivalence class $E$, instead of returning the set $E$ we can return an output $o \in E$ selected uniformly at random. We define this enforcement as follows:

$$\textsc{EnfRnd}(\pi, \xi)(o \mid i) = \frac{1}{|[o]_\xi|} \textsc{Enf}(\pi, \xi)([o]_\xi \mid i)$$

We illustrate $\textsc{EnfRnd}(\pi, \xi)$ for the program $\pi$ and enforcement $\xi$ of our motivating example in Figure 4(c).

We remark that both ways of enforcing $\xi$ are equivalent from a security point of view: for any program $\pi$, enforcement $\xi$ for $\pi$, attacker belief $\delta$, and privacy policy $\Phi$, we have $\textsc{Enf}(\pi, \xi), \delta \models \Phi$ if and only if $\textsc{EnfRnd}(\pi, \xi), \delta \models \Phi$.

We say that $\xi$ *enforces* a privacy policy $\Phi$ for a program $\pi$ and attacker belief $\delta$ if $\textsc{Enf}(\pi, \xi), \delta \models \Phi$. Note that for any program $\pi$, attacker belief $\delta$, and privacy policy $\Phi$, if $\pi, \delta \models \Phi$ then for any enforcement $\xi$ for $\pi$ we have $\textsc{Enf}(\pi, \xi), \delta \models \Phi$.

Finally, we remark that our notion of enforcement is complete. Suppose we have a program $\pi$, an attacker belief $\delta$, and a privacy policy $\Phi$ such that $\pi, \delta \not\models \Phi$. Then, there exists a program $\pi'$ such that $\pi', \delta \models \Phi$ if and only if there exists an enforcement $\xi$ for $\pi$ such that $\textsc{Enf}(\pi, \xi), \delta \models \Phi$. This means that if no equivalence relation enforces the privacy policy for the given program and attacker belief, then no program satisfies the policy for the given attacker. We prove this statement in Section 4.4.2.

## 4.2 Optimality of enforcement

Whenever there exists some enforcement $\xi$ for a program and a policy, then the enforcement $\xi_\perp$ which conflates all outputs into one is also a valid one. Hence we define the notion of enforcement optimality to be able to synthesize the best feasible enforcement. Formally, an optimality order is a total preorder on the set of enforcements, i.e. $(\mathcal{E}(O), \leq)$, where by $\mathcal{E}(O)$, we denote the set of equivalence relations over $O$ and $\leq$ is a total preorder on $\mathcal{E}(O)$ (the relation $\leq$ has to be reflexive, transitive, and total).

*4.2.1 Permissiveness.* A prime instance of an optimality order is the *permissiveness*. For an enforcement $\xi$, we define its permissiveness as $|O/\xi|$, i.e. the number of equivalence classes of $\xi$. For a program $\pi$, we obtain a total preorder on the permissiveness of the enforcements for $\pi$, where is $\xi$ at least as permissive as $\xi'$ if $|O/\xi| \geq |O/\xi'|$. The least-permissive enforcement for a program $\pi$ is the relation $\xi_\perp = O \times O$. The program $\textsc{Enf}(\pi, \xi_\perp)$ can be seen as the program that always returns a deny decision. This is because $\textsc{Enf}(\pi, \xi_\perp)$ always returns $O$ and the attacker's revised belief equals her prior belief. Conversely, the most-permissive enforcement for $\pi$ is $\xi_\top = \{(o, o) \mid o \in O\}$. The program $\textsc{Enf}(\pi, \xi_\top)$ can be seen as identical to $\pi$ as it always outputs singleton equivalence classes that contain the output returned by $\pi$. In fact, using the semantics of $\textsc{EnfRnd}$, we get $\textsc{EnfRnd}(\pi, \xi_\top) = \pi$.

*4.2.2 Answer precision.* Another useful example of optimality is *answer precision*, i.e. the number of singletons in an equivalence relation, given by $\mathrm{Sing}(\xi) = |\{o \in O \mid |[o]_\xi| = 1\}|$. We define $\xi \geq \xi'$ iff $\mathrm{Sing}(\xi) \geq \mathrm{Sing}(\xi')$, i.e. $\xi$ has at least as many singleton classes as $\xi'$. The set of least elements is given by $[\xi_\perp]_\leq$ and it contains all the equivalence classes having zero singletons. By answer precision, all these enforcements are considered equally imprecise, since for all of them there is never an exact answer (a singleton class). On the other hand, the enforcement $\xi_\top$ is the unique most-precise enforcement, since it is the only one that always gives an exact answer.

***Optimal Enforcement.*** An enforcement $\xi$ is *optimal* for a program $\pi$, an attacker belief $\delta$, and a privacy policy $\Phi$ with respect to an order $\leq$ if $\textsc{Enf}(\pi, \xi), \delta \models \Phi$ and for any enforcement $\xi'$ such that $\textsc{Enf}(\pi, \xi'), \delta \models \Phi$ we have $\xi' \leq \xi$.

To synthesize an optimal enforcement with regard to a partial order (a more general notion), we can use the total preorder given by the height of each element in the said poset. When we compute an optimal enforcement with regard to the height total preorder, the enforcement is also optimal with regard to the preorder. For example, the permissiveness objective also guarantees optimality with regard to the partial order given by set inclusion (i.e. $E_1$ is above $E_2$ iff $E_1 \subseteq E_2$).

## 4.3 Synthesis Problem

We now define optimal privacy enforcement synthesis:

*Definition 4.1.* The *optimal privacy enforcement* synthesis problem is defined as follows:

**Input.** A probabilistic program $\pi$, an attacker belief $\delta$, a privacy policy $\Phi$, and an optimality order $\leq$.

**Output.** An optimal enforcement $\xi$ for $\pi$, $\delta$, $\Phi$, and $\leq$ if such an enforcement exists; otherwise return unsat.

We remark on several key points. First, our synthesis problem is decidable since *(i)* there are finitely many enforcements $\xi$ for any program with finitely many inputs $\mathcal{I}$ and outputs $O$ and *(ii)* checking $\textsc{Enf}(\pi, \xi), \delta \models \Phi$ is decidable. For some notions of optimality, like permissiveness, the synthesis problem is, however, NP-hard, as we prove in Section 4.4. Second, we can check whether the synthesis problem returns an enforcement $\xi$ or unsat by checking whether the attacker belief $\delta$ about the predicates in $\Phi$ are within their corresponding bounds. Finally, for both the permissiveness and answer precision, if $\pi, \delta \models \Phi$, then the synthesis problem returns the most-permissive enforcement $\xi_\top = \{(o, o) \mid o \in O\}$ for $\pi$. The synthesized enforcement thus does not unnecessarily change the semantics of $\pi$ whenever $\pi$ already satisfies the policy for the given attacker belief.

We remark that a solution to the synthesis problem can be used to provide guarantees for a set of attacker beliefs $\{\delta_1, .., \delta_n\}$. First, we synthesize an enforcement $\xi_i$ for each attacker belief $\delta_i$. Then, we take the union of the enforcements $\xi = \xi_1 \cup .. \cup \xi_n$ and return the transitive closure $\xi^*$ of $\xi$. The synthesized enforcement $\xi^*$ is guaranteed to satisfy the policy for all attacker beliefs $\delta_1, \ldots, \delta_n$.

## 4.4 Complexity and Completeness

In this subsection, we give the results on the complexity of optimal privacy enforcement synthesis problem and show that our notion of enforcement is complete.

*4.4.1 Complexity.* For the case of permissiveness optimality, the optimal enforcement problem is NP-equivalent, as stated in the following theorem.

Theorem 4.2. *The optimal privacy enforcement synthesis problem is* NP*-equivalent (*NP*-hard and* NP*-easy) for permissiveness.*

As the theorem states, for the case of permissiveness, the problem is NP-hard even for synthesis instances with singleton policies, i.e. policies that contain only one security assertion. In Appendix A, we prove that the problem is NP-hard by reducing the partition problem to it, and we show that it is NP-easy by giving an NP-oracle polynomial Turing machine that solves it.

For the case of answer precision, the synthesis problem can be solved in polynomial time for instances with singleton policies (i.e., having one security assertion).

Theorem 4.3. *For instances with singleton policies, the optimal privacy enforcement synthesis problem is in* PTIME *for answer precision optimality.*

The proof of this theorem can be found in Appendix A.4, where we give a polynomial-time algorithm that produces optimal answer-precision enforcements for singleton policies. It is an open problem

**Algorithm 1:** The algorithm SynSMT$(\pi, \delta, \Phi)$

**Input**: A probabilistic program $\pi$ with outputs $O$, an attacker belief $\delta$, a privacy policy $\Phi$, and an objective function $\Psi_{obj}$.
**Output**: An optimal enforcement mechanism $\xi$ such that $\text{Enf}(\pi, \xi), \delta \models \Phi$ if one exists, otherwise unsat.

```
1 begin
2     ψ_assert ← Assert(π, δ, Φ)
3     if IsSat(ψ_assert) then
4         M ← Max(ψ_assert, ψ_obj)
5         return ker(M)
6     else
7         return unsat
```

whether for answer precision the synthesis problem is solvable in polynomial time for the general case (i.e., for policies with multiple security assertions). We conjecture that it is, since our greedy algorithm SynGrd, presented in Section 6, produces optimal enforcements for all our benchmarks.

*4.4.2 Completeness.* Our notion of enforcement is complete: for a program $\pi$, a prior belief $\delta$, and a policy $\Phi$, there exists a valid enforcement $\xi$ ($\xi$ is valid if $\delta, \text{Enf}(\pi, \xi) \models \Phi$) if and only if there exists some arbitrary program $\pi'$ satisfying the policy $\Phi$ for $\delta$, i.e. $\pi', \delta \models \Phi$. In other words, in the case no valid enforcement exists, it is not by a shortage of enforcement equivalent relations, but because of the attacker's prior belief $\delta$ and the policy $\Phi$ that cannot be satisfied by *any* program.

Theorem 4.4. *Let $\mathcal{I}$ be a set of inputs and $O$ a set of outputs, $\delta \in \mathcal{D}(\mathcal{I})$ an attacker belief, and $\Phi$ a privacy policy. There exists a program $\pi \in \mathbb{M}_{\mathcal{I},O}$ such that $\pi, \delta \models \Phi$ if and only if for any program $\pi' \in \mathbb{M}_{\mathcal{I},O}$ we have that $\text{Enf}(\pi', \xi_\perp) \models \Phi$.*

The proof of this is in Appendix A.

## 5 SMT-BASED SYNTHESIS ALGORITHM

We now present our synthesis algorithm SynSMT.

***High-level Idea.*** SynSMT takes as input a probabilistic program $\pi \in \mathbb{M}_{\mathcal{I}\times O}$ defined over inputs $\mathcal{I}$ and outputs $O$, an attacker belief $\delta \in \mathcal{D}(\mathcal{I})$, and a privacy policy $\Phi$. It is based on two key insights. First, we represent the search space of possible enforcements $\xi \in O \times O$ with integer variables $C_1, \ldots, C_{|O|}$, and then encode the satisfaction of $\text{Enf}(\pi, \xi), \delta \models \Phi$ into SMT constraints over these variables. The encoding guarantees that any model of the SMT constraints identifies an enforcement $\xi$ such that $\text{Enf}(\pi, \xi) \models \Phi$. Second, we encode the optimality ordering over enforcements as an objective function that returns the rank of an enforcement (e.g. the number of equivalence classes, or the number of singletons) and use this function as an optimization goal.

***Key Steps.*** We introduce an integer variable $C_i$ for each output $o_i \in O$ to encode all possible enforcements. Each $C_i$ is assigned a value from $\{1, \ldots, |O|\}$ that represents the equivalence class to which $o_i$ belongs; e.g., if only $C_i$ and $C_j$ are set to $k$, then the equivalence class $E_k$ is $\{o_i, o_j\}$. A model is a mapping $\mathcal{M} : \{C_1, \ldots, C_{|O|}\} \to \{1, \ldots, |O|\}$ that assigns an integer value between 1 and $|O|$ to each variable $C_i$. The kernel of a model $\mathcal{M}$ is defined as the set $\ker(\mathcal{M}) = \{(o_i, o_j) \in O \times O \mid \mathcal{M}(C_i) = \mathcal{M}(C_j)\}$. Hence a model $\mathcal{M}$ identifies the enforcement $\xi = \ker(\mathcal{M})$.

$$\text{Assert}(\pi, \delta, \Phi) := \psi_{\text{range}} \wedge \psi_{\text{bounds}}$$

$$\psi_{\text{range}} \equiv \bigwedge_{i=1}^{|O|} C_i \geq 1 \wedge C_i \leq |O|$$

$$\psi_{\text{bounds}} \equiv \bigwedge_{\ell=1}^{|\Phi|} \bigwedge_{j=1}^{|O|} \psi^j_{\text{non-empty}} \Rightarrow p^j_\ell \in [a_\ell, b_\ell]$$

$$\psi^j_{\text{non-empty}} \equiv \bigvee_{i=1}^{|O|} C_i = j$$

$$p^j_\ell = \frac{\sum_{i=1}^{|O|} [C_i = j] \cdot \mathbb{P}^\pi_\delta(I \in S_\ell \mid O = o_i) \cdot \mathbb{P}^\pi_\delta(O = o_i)}{\sum_{i=1}^{|O|} [C_i = j] \cdot \mathbb{P}^\pi_\delta(O = o_i)}$$

$$\text{Obj}_{\text{cls}}(n) := \textbf{maximize} \quad \textstyle\sum_{j=1}^{|O|} \left[\psi^j_{\text{non-empty}}\right]$$

$$\text{Obj}_{\text{sing}}(n) := \textbf{maximize} \quad \textstyle\sum_{j=1}^{|O|} \left[\left(\sum_{i=1}^{|O|} [C_j = i]\right) = 1\right]$$

**Figure 5: SMT constraints $\text{Assert}(\pi, b, \Phi)$ and two objective functions ($\text{Obj}_{\text{cls}}(n)$ for permissiveness and $\text{Obj}_{\text{sing}}(n)$ for answer precision) for a synthesis instance with a program $\pi$, with outputs $O = \{o_1, \ldots, o_n\}$, attacker belief $\delta$, and privacy policy $\Phi = \{(S_1, [a_1, b_1]), \ldots, (S_k, [a_k, b_k])\}$. The variables in the constraints are $\{C_1, \ldots, C_n\}$ of type integer.**

We show the main steps of SynSMT in Algorithm 1. At Line 2, the algorithm generates the SMT constraint $\psi_{\text{assert}}$ using the formula $\text{Assert}(\pi, \delta, \Phi)$, as defined in Figure 5. The two probabilities $\mathbb{P}^p_\delta(I \in S_\ell \mid O = o_i)$ and $\mathbb{P}^p_\delta(O = o_i)$ that appear in the SMT constraints are constants that are computed from the program $\pi$ and attacker belief $\delta$ beforehand using a probabilistic solver. We detail this step in Section 7. The constraint $\psi_{\text{assert}}$ conjoins $\psi_{\text{range}}$ and $\psi_{\text{bounds}}$. The constraint $\psi_{\text{range}}$ encodes that the range of all variables $C_i$ is $\{1, \ldots, |O|\}$, and $\psi_{\text{bounds}}$ encodes that for each belief bound $(S_\ell, [a_\ell, b_\ell]) \in \Phi$, we have $\text{Enf}(\pi, \xi), \delta \models (S_\ell, [a_\ell, b_\ell])$, where $\xi$ is the enforcement identified by the variables $C_1, \ldots, C_{|O|}$.

We use $p^j_\ell$ to denote the attacker belief about predicate $S_\ell$ after observing the equivalence class $E_j = \{o_i \in O \mid C_i = j\}$. This is defined as $p^j_\ell = \mathbb{P}^p_\delta(I \in S_\ell \mid O \in E_j)$ and is computed by summing over all outputs that belong to $E_j$. We use Iverson bracket notation $[\psi]$ to denote the function that returns 1 if $\psi$ holds, and 0 otherwise. For any non-empty equivalence class $E_j$, the value of $p^j_\ell$ must be within $[a_\ell, b_\ell]$ as defined by the belief bound $(S_\ell, [a_\ell, b_\ell])$. The disjunction $\psi^j_{\text{non-empty}}$ encodes whether $E_j$ is non-empty.

SynSMT receives the objective function $\psi_{\text{obj}}$ as an input. Examples of objective functions are given in Figure 5. The function $\text{Obj}_{\text{cls}}(n)$ maximizes the sum of all non-empty equivalence classes, i.e. it maximizes the permissiveness of the enforcement.

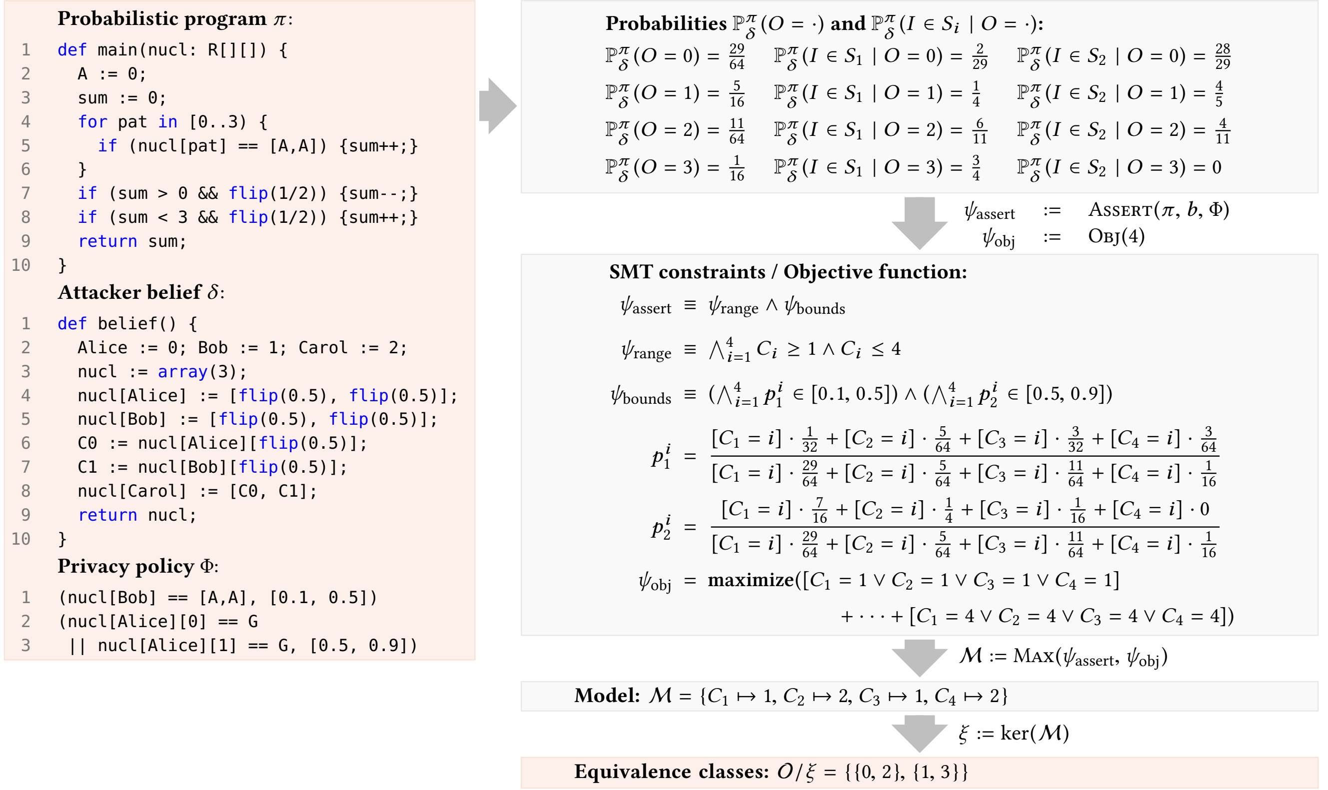



**Figure 6: Steps of the algorithm SynSMT for Example 1: The left box depicts the input, the bottom right the output, and the gray boxes depict intermediate computation steps of the algorithm.**

The function $\textsc{Obj}_{\text{sing}}(n)$ maximizes the count of all singleton classes, i.e. it maximizes the answer precision.

To check whether there exists an enforcement $\xi$ such that we have $\textsc{Enf}(\pi, \xi), b \models \Phi$, the algorithm calls $\textsc{IsSat}(\psi_{\text{assert}})$, which returns whether $\psi_{\text{assert}}$ is satisfiable. If $\psi_{\text{assert}}$ is unsatisfiable, then it simply returns unsat. Otherwise, SynSMT calls the procedure $\textsc{Max}(\psi_{\text{assert}}, \psi_{\text{obj}})$, which returns a model $\mathcal{M}$ of the SMT constraint $\psi_{\text{assert}}$ that maximizes the objective function $\psi_{\text{obj}}$. Finally, it returns $\ker(\mathcal{M})$ which defines the enforcement identified by the variables $C_1, \ldots, C_{|O|}$.

*Example 5.1.* We consider the same scenario as the one in our motivating example; see Section 2.1. The input to SynSMT is given in Figure 6 (left). The program $\pi$ returns the number of patients with two adenine ($A$) nucleotides and randomly adds $\pm 1$ to the result. Since there are three patients, the set of outputs for $\pi$ is $O = \{0, 1, 2, 3\}$.

The attacker belief is defined by `belief()`. In contrast to the belief of Figure 2(b), here we assume that the frequency of guanine is 0.5 to simplify the fractions in our example.

The policy $\Phi$ states that according to the attacker belief: *($S_1$)* the probability that Bob's nucleotides are $AA$ is between 0.1 and 0.5, *($S_2$)* the probability that Alice has a guanine nucleotide is between 0.5 and 0.9.

We now describe the intermediate steps of SynSMT, depicted in the gray boxes in Figure 6. First, we compute the probabilities $\mathbb{P}^\pi_\delta(O = o)$ and $\mathbb{P}^\pi_\delta(I \in S_i \mid O = o)$ for each output $o \in \{0, \ldots, 3\}$ and belief bound $i \in \{1, 2\}$.

Next, SynSMT generates the SMT constraint $\psi_{\text{assert}}$ and the objective function $\psi_{\text{obj}}$ (chosen as $\textsc{Obj}_{\text{cls}}(n)$ for this example to optimize permissiveness), over the integer variables $C_1$, $C_2$, $C_3$, and $C_4$. The constraints $\psi_{\text{range}}$ restricts the range of all the variables to $\{1, 2, 3, 4\}$, and $\psi_{\text{bounds}}$ restricts the probability of the predicates $S_1$ and $S_2$ in all equivalence classes to be in $[0.1, 0.5]$ and $[0.5, 0.9]$, respectively. The formulas $p_1^i$ and $p_2^i$ are symbolically defined as described in Figure 5.

A model of $\psi_{\text{assert}}$ that maximizes the objective $\psi_{\text{obj}}$ is $\mathcal{M} = \{C_1 \mapsto 1, C_2 \mapsto 2, C_3 \mapsto 1, C_4 \mapsto 2\}$. Hence, the synthesized enforcement is $O/\ker(\mathcal{M}) = \{\{0, 2\}, \{1, 3\}\}$. ■

Finally, we state the correctness of our algorithm.

Theorem 5.2. *Let $\pi$ be a probabilistic program, $\delta$ an attacker belief, and $\Phi$ a privacy policy. If there is no enforcement $\xi$ such that $\textsc{Enf}(\pi, \xi), \delta \models \Phi$, then $\textsc{SynSMT}(\pi, \delta, \Phi) =$ unsat. Otherwise, $\textsc{SynSMT}(\pi, \delta, \Phi) = \xi$ such that $\xi$ is an optimally permissive enforcement for $\pi$, $\delta$, and $\Phi$.*

The proof of this theorem is in Appendix A.

## 6 EFFICIENT GREEDY SYNTHESIS ALGORITHM

We present our algorithm SynGrd, which produces correct enforcements but does not guarantee that they are optimal.

***High-level Idea.*** When optimizing for permissiveness, SynGrd starts with the most refined equivalence relation $\xi_\top$, which has one equivalence class per output, and then iteratively joins equivalence classes until the relation enforces the policy for the given attacker. When optimizing for answer precision, we start with an equivalence relation that joins all the outputs that violate the policy. Then, in each step, the algorithm selects an equivalence class $E$ that, if output to the attacker, would violate the policy. The class $E$ is then joined with another equivalence class $E'$ such that the revised attacker belief about the predicates defined in the policy after observing the new equivalence class $E \cup E'$ is *closer* to the bounds defined in the policy. We detail these steps below.

***Notation.*** We use the following notation. Let the policy be $\Phi = \{(S_1, [a_1, b_1]), \ldots, (S_k, [a_k, b_k])\}$ with $k = |\Phi|$. Given an equivalence class $X \subseteq O$, we define the $k$-dimensional vector $\vec{p}_X^{\,\delta,\pi} = (\mathbb{P}_\delta^\pi(I \in S_i \mid O \in X))_{i=1}^k$, where the $i$-th element is the attacker belief about the predicate $S_i$ given that the program returns equivalence class $X$. Further, we define the $k$-dimensional box $\vec{\alpha}_\Phi = ([a_i, b_i])_{i=1}^k$, where $[a_i, b_i]$ are the bounds defined by the belief bound $(S_i, [a_i, b_i]) \in \Phi$. We write $\vec{p}_X^{\,\delta,\pi} \in \vec{\alpha}_\Phi$ to denote that for each belief bound $(S_i, [a_i, b_i])$, we have $\mathbb{P}_\delta^\pi(I \in S_i \mid O \in X) \in [a_i, b_i]$. An equivalence relation $\xi$ enforces the privacy policy $\Phi$ for the given $\pi$ and $\delta$ if for each equivalence class $E \in O/\xi$, we have $\vec{p}_E^{\,\delta,\pi} \in \vec{\alpha}_\Phi$.

In Figure 7, we depict the 2-dimensional box defined by the privacy policy of Example 5.1. The $X$-axis shows the probability of the predicate $S_1$ and the $Y$-axis that of $S_2$. The gray area depicts the box $\vec{\alpha}_\Phi$, i.e. all vectors that lie within the two belief bounds in $\Phi$. We depict the vectors $\vec{p}_X^{\,\delta,\pi}$ for all singleton equivalence classes $X \in \xi_\top$, as well as those for classes $\{0, 2\}$ and $\{1, 3\}$, which are produced by the algorithm. We have $\vec{p}_{\{0\}}^{\,\delta,\pi} \notin \vec{\alpha}_\Phi$, as this vector lies outside the box $\vec{\alpha}_\Phi$, and thus violates at least one belief bound in $\Phi$. In contrast, we have $\vec{p}_{\{0,2\}}^{\,\delta,\pi} \in \vec{\alpha}_\Phi$, and so the class $\{0, 2\}$ satisfies all belief bounds.

***Key Steps.*** The main steps of SynGrd are given in Algorithm 2. At Line 2, the algorithm checks whether the condition $\bigvee_{i=1}^k \mathbb{P}_\delta^\pi(I \in S_i) \notin [a_i, b_i]$ holds. The satisfaction of this condition implies the non-existence of an equivalence relation that enforces $\Phi$ for the given $\pi$ and $\delta$. If no such enforcement exists, the algorithm returns unsat.

At Line 5, the algorithm constructs the initial equivalence relation. For the permissiveness goal, the most refined equivalence relation is used. For the answer precision goal, the equivalence relation joining all the violating outputs into one class $C$ and keeping the rest of outputs, that do not violate the policy, as singletons is used. The rationale is that these outputs would never be singletons in a valid enforcement.

The algorithm then iteratively performs the following steps until the constructed equivalence relation $\xi$ enforces the policy (checked at Line 9):

**Algorithm 2:** The algorithm SynGrd$(\pi, \delta, \Phi)$

```
   Input: A probabilistic program π, an attacker belief δ, a policy
          Φ = {(S_1, [a_1, b_1]), ..., (S_k, [a_k, b_k])}, and an
          optimization goal
          goal ∈ {"permissiveness", "precision"}
   Output: An equivalence relation ξ enforcing the policy.
 1 begin
 2   if ⋁_{i=1}^k P_δ^π(I ∈ S_i) ∉ [a_i, b_i] then
 3     return unsat
 4   if goal = "permissiveness" then
 5     ξ ← {(o, o) | o ∈ O}
 6   else if goal = "precision" then
 7     C ← {o ∈ O | p_{o}^{δ,π} ∉ α_Φ}
 8     ξ ← {(o, o) | o ∈ O} ∪ C × C
 9   while ∃E ∈ O/ξ : p_E^{δ,π} ∉ α_Φ do
10     E ← arg max_{E∈O/ξ} Dist(p_E^{δ,π}, α_Φ)
11     if ∃E' ∈ O/ξ : p_{E∪E'}^{δ,π} ∈ α_Φ then
12       E' ← arg min_{E'∈O/ξ} ||p_{E∪E'}^{δ,π} − ((a_i+b_i)/2)_{i=1}^k||
13     else
14       E' ← arg min_{E'∈O/ξ} Dist(p_{E∪E'}^{δ,π}, α_Φ)
15     ξ ← ξ ∪ {(e, e') | e ∈ E ∧ e' ∈ E'}
16   return ξ
```

***Pick Most Violating Class.*** At Line 10, the algorithm picks the class $E$ with the greatest distance between the vector $\vec{p}_E^{\,\delta,\pi}$ and the box $\vec{\alpha}_\Phi$, which is defined as the distance between the point $\vec{p}_X^{\,\delta,\pi}$ and the closest point $\vec{q}$ inside the box $\vec{\alpha}_\Phi$:

$$\textsc{Dist}(\vec{p}_X^{\,\delta,\pi}, \vec{\alpha}_\Phi) = \min_{\vec{q} \in \vec{\alpha}_\Phi} \|\vec{p}_X^{\,\delta,\pi} - \vec{q}\|$$

***Pick Merge Candidate.*** At Line 11, SynGrd checks if there is a class $E'$ such that if merged with $E$, the resulting vector $\vec{p}_{E\cup E'}^{\,\delta,\pi}$ is in the box $\vec{\alpha}_\Phi$. If this is the case, then SynGrd picks the class $E'$ that has the point $\vec{p}_{E\cup E'}^{\,\delta,\pi}$ closest to the center of $\vec{\alpha}_\Phi$ (Line 12). Otherwise, SynGrd picks the class $E'$ that has the point $\vec{p}_{E\cup E'}^{\,\delta,\pi}$ closest to the box $\vec{\alpha}_\Phi$. (Line 14)

Note that SynGrd prefers to pick a class $E'$ such that the merged class $E \cup E'$ is in box $\vec{\alpha}_\Phi$, as this implies that the merged class need not be further merged with other classes.

***Merge.*** At Line 15, the algorithm combines the class $E$ and the picked class $E'$.

The loop terminates when the equivalence relation $\xi$ enforces the policy for the given program and attacker belief. At this point SynGrd returns $\xi$.

*Example 6.1.* In Figure 7, we graphically illustrate the steps of SynGrd on Example 5.1 for the permissiveness goal. The vectors $\vec{p}_{\{0\}}, \vec{p}_{\{1\}}, \vec{p}_{\{2\}}, \vec{p}_{\{3\}}$ denote the probabilities of the predicates $S_1$ and $S_2$ for all four singleton equivalence classes. The while loop's condition evaluates to true since the vectors $\vec{p}_{\{0\}}^{\,\delta,\pi}$, $\vec{p}_{\{2\}}^{\,\delta,\pi}$, and $\vec{p}_{\{3\}}^{\,\delta,\pi}$

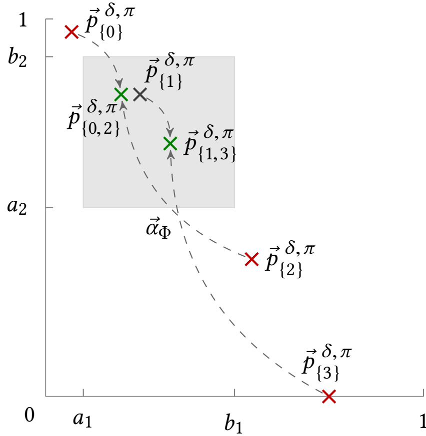



**Figure 7: Equivalence classes computed by the greedy synthesis algorithm SynGrd for our Example 5.1 for the permissiveness goal.**

lie outside the box $\vec{\alpha}_\Phi$. At Line 10, SynGrd selects $\{3\}$ since $\vec{p}^{\,\delta,\pi}_{\{3\}}$ is the most distant vector from $\vec{\alpha}_\Phi$. SynGrd merges $\{3\}$ with $\{1\}$ as the vector $\vec{p}^{\,\delta,\pi}_{\{1,3\}}$ is closest to the center of $\vec{\alpha}_\Phi$. In the second iteration of the while loop, SynGrd merges the equivalence classes $\{0\}$ and $\{2\}$. The algorithm returns the equivalence classes $\{\{0,2\},\{1,3\}\}$. ■

***Running Time.*** The running time of SynGrd is $O(|O|^2)$. The while loop is executed at most $|O|$ times, since initially we have $|O/\xi| = |O|$, and each iteration it decreases by 1. When it reaches 1, the while loop's condition must be satisfied. All expressions in the while loop's body can be evaluated in $O(|O|)$ when $\xi$ is represented as $O/\xi$.

Theorem 6.2. *Let $\pi$ be an arbitrary probabilistic program, $\delta$ an arbitrary attacker belief, and $\Phi$ an arbitrary privacy policy. If there is no equivalence relation $\xi$ such that $\textsc{Enf}(\pi,\xi),\delta \models \Phi$, then $\textsc{SynGrd}(\pi,\delta,\Phi) =$ unsat. Otherwise, we have $\textsc{SynGrd}(\pi,\delta,\Phi) = \xi$ and $\textsc{Enf}(\pi,\xi),\delta \models \Phi$.*

The proof of this theorem is in Appendix A.

# 7 IMPLEMENTATION

We now describe the Spire system, an end-to-end implementation of our enforcement synthesis approach.

***Inputs.*** Spire takes three files as input: *(i)* a probabilistic program, *(ii)* a probabilistic program that encodes an attacker belief, and *(iii)* a text file that defines a privacy policy. Inputs *(i)* and *(ii)* are specified in the Psi language [22]. The Psi language, presented in Appendix B, is an imperative probabilistic language that operates on real-valued scalar and array data, and supports probabilistic assignments, observe statements, as well as the standard sequence, conditional, and bounded loop statements. For more details on the Psi language see [22]. Input *(iii)* is in the format $(Expr, a, b)$ where $Expr$ is a Psi expression and $a$ and $b$ are bounds. In addition, a parameter is passed defining whether to optimize for permissiveness or answer precision. Note that other notions of optimality can be easily added by specifying a custom objective function.

***Bayesian Inference.*** We use the Psi solver [22] to perform symbolic inference. For each belief bound $(S, [a,b])$, Spire calls the Psi solver to compute a symbolic expression that captures the probability distribution $\mathbb{P}^\pi_\delta(I \in S \mid O = x)$, where $x$ is a symbolic variable. This step is $\#P$-complete [55]. Note that the symbolic expression captures the probability of the predicate $S$ for all possible outputs. This expression is specified in the SMT-LIBv2 format [4]. We evaluate this expression using off-the-shelf SMT solvers, for all possible outputs, to obtain the probabilities used by the synthesis algorithms. Similarly, we use the Psi solver to compute a symbolic expression that captures the probability distribution $\mathbb{P}^\pi_\delta(O = y)$. These distributions are sufficient to derive all necessary probabilities used by the two synthesis algorithms.

***Synthesis Algorithms.*** Spire implements the two synthesis algorithms in C# (in roughly 2.5K LOC). In the implementation of the SynSMT algorithm, Spire calls the Z3 SMT solver[8, 16] to solve linear optimization problems over SMT constraints; see Line 4 of Algorithm 1.

Spire supports an interactive mode, where the attacker may ask to run a program multiple times. In this mode, the initial attacker belief is the one provided as input to Spire. For subsequent iterations, Spire keeps track of the revised attacker belief using symbolic inference. Concretely, Spire uses Psi to compute a symbolic expression over a variable $x$ that captures the attacker belief $\mathbb{P}^\pi_\delta(I = x)$.

***Output.*** As output, Spire returns the equivalence classes computed by the two synthesis algorithms. It also outputs the encoding of the enforcement in the Psi language, as illustrated in Lines 9-15 in Figure 3.

# 8 EVALUATION

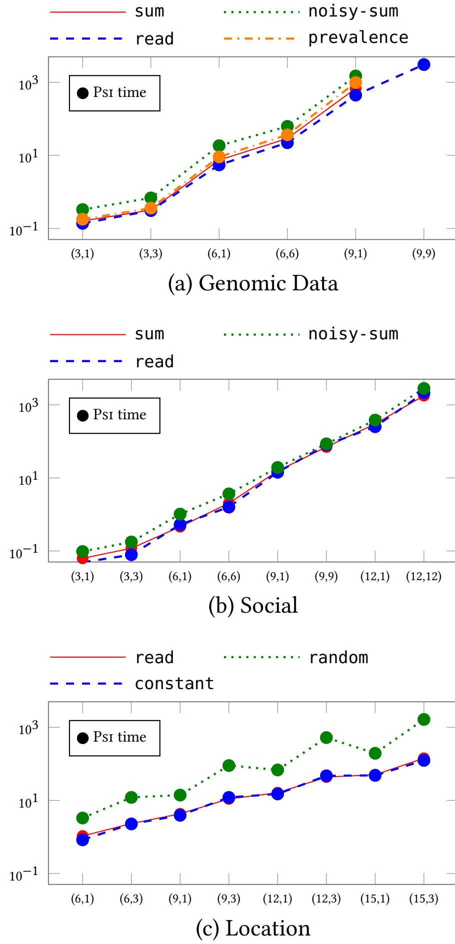
In this section, we evaluate our Spire system as follows: *(i)* we compare the algorithms SynGrd and SynSMT, *(ii)* we evaluate Spire's performance, and *(iii)* we measure the permissiveness and answer precision of the synthesized enforcements. We first describe our experiments and then report and discuss our results.

## 8.1 Experiments

We perform experiments on instances of 10 different programs from 3 scenarios adopted from the literature. For each scenario, we have both deterministic, and probabilistic programs. Here, we briefly sketch the scenarios. Full details can be found in Appendix C.

***Genomic Data.*** This scenario is the same as the one described in Section 2, but here we scale the number of patients and the policy size. We experiment with four programs: `sum` given in Figure 2(a), `noisy-sum` which is like `sum` but randomly adds ±1 as in Figure 6, `read` returns the nucleotides of a patient, and `prevalence` returns the number of patients with `AA` nucleotides. We generate synthesis instances $(n, m)$ where $n$ is the number of patients and $m$ the size of the privacy policy.

***Social.*** This scenario, borrowed from [23], models a social network where users express their political affiliations, which are correlated based on the friendship relations [29, 35, 59]. We experiment with three programs: `sum` returns the number of users affiliated with a

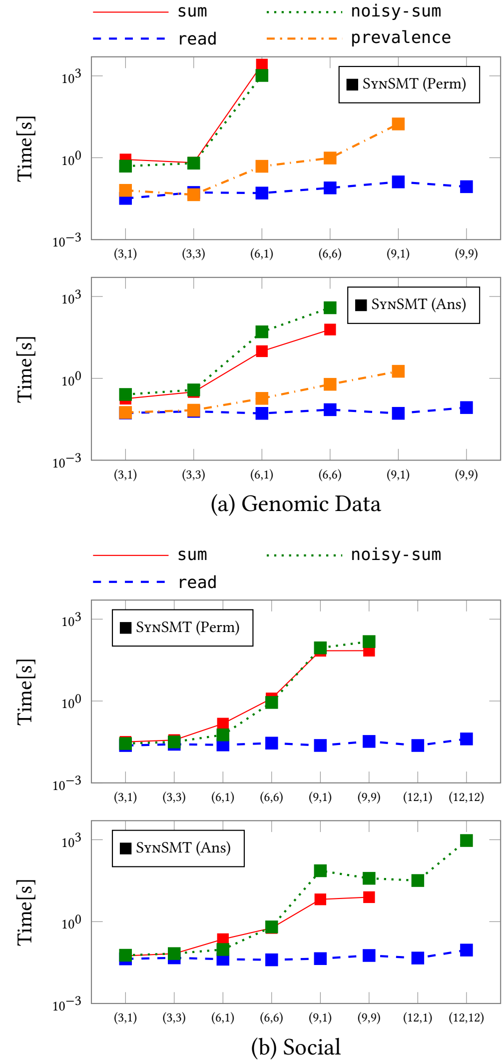


(a) Genomic Data

(b) Social

(c) Location

**Figure 8: Running times of Psi for Bayesian inference.**



(a) Genomic Data

(b) Social

**Figure 9: Running times of the synthesis algorithm SynSMT. For both (a) and (b) we present two plots: the first shows running times of SynSMT optimizing for permissiveness and the second SynSMT optimizing for answer precision.**

particular party, `noisy-sum` is like `sum` but randomly adds ±1, and `read` returns the political affiliation of a user. We generate synthesis instances $(n, m)$ where $n$ is the number of users and $m$ the privacy policy size. The policy bounds the attacker belief about a user's affiliation.

***Location.*** This scenario models a user protecting his location [34, 52]. We borrow three programs from [48]: `read` returns the user's location, `constant` and `random` return a constant and, respectively, random coordinate if the user is within a sensitive area. We generate synthesis instances $(n, m)$ where $n$ is the width and height of a rectangular grid and $m$ the size of the privacy policy. The policy bounds the attacker belief about the user being in a sensitive location.

## 8.2 Results

We ran all synthesis instances with each algorithm SynSMT and SynGrd, optimizing for permissiveness and answer precision with both algorithms. We used a 32-core machine with four 2.13GHz Xeon processors running Ubuntu Linux 16.04. We set a timeout of 60 minutes for the symbolic inference using Psi, SynSMT, and SynGrd, separately for each. All numbers reported below are averaged over 10 runs.

*8.2.1 SynSMT vs. SynGrd.* To compare the two synthesis algorithms, we measure: (i) the percentage of instances where SynGrd produces an optimal enforcement with respect to the permissiveness and the answer precision optimality orders, and (ii) how close to the optimal is the enforcement synthesized by SynGrd. We can measure these whenever SynSMT terminates within the timeout, since the synthesized enforcement is guaranteed to be optimal.

***Permissiveness.*** The percentages of instances where SynGrd produces an optimal enforcement with respect to permissiveness measured across all instances of the medical and social scenarios are

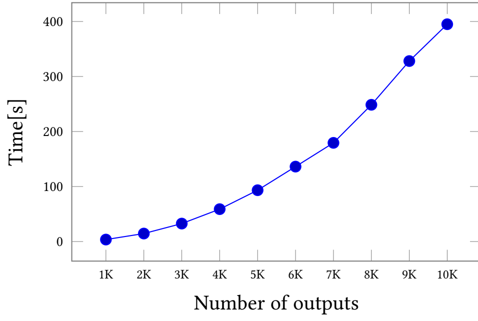
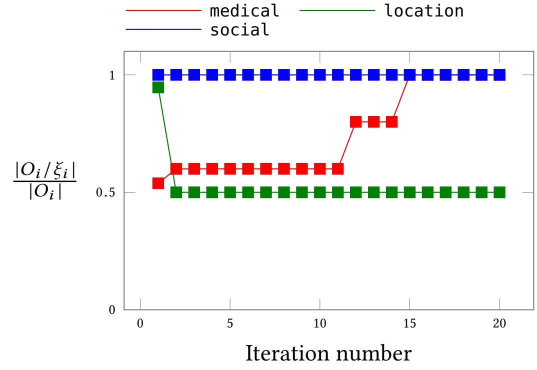
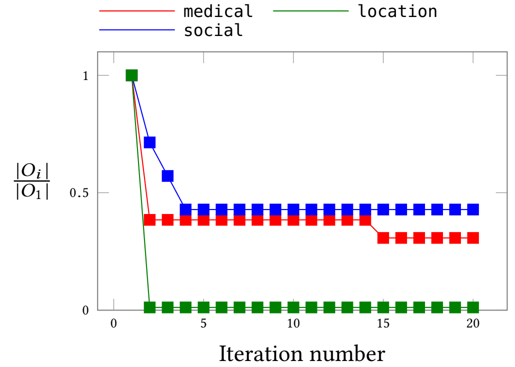



**Figure 10: Running times of SynGrd for large inputs.**




a) Ratio of classes count to outputs count for each iteration




b) Number of outputs with non-zero probability in each iteration as a ratio of the total number of outputs

**Figure 11: Permissiveness for Interactive Attackers**

82.35% and 95%, respectively. We do not know this percentage for the location scenario instances because SynSMT did not terminate.

To measure how close to the optimal enforcement the enforcements synthesized by SynGrd are with respect to permissiveness, we measured $\frac{|O/\xi_{\text{SynGrd}}|}{|O/\xi_{\text{SynSMT}}|}$ for each instance, where $\xi_{\text{SynGrd}}$ and $\xi_{\text{SynSMT}}$ are the enforcements synthesized by SynGrd and SynSMT, respectively. This indicates the amount of classes given by SynGrd as a fraction of the optimal enforcement. The average percentages across all instances of the medical and social scenarios are 91.11% and 98.33%, respectively. As before, we could not measure this number for the location scenario as SynSMT timed out for all location instances.

Overall, these three metrics indicate that for our examples SynGrd often produced an optimal enforcement with respect to permissiveness or an enforcement that is close to the optimum.

***Answer precision.*** For all synthesis instances where SynSMT produced an enforcement within the timeout, SynGrd produced an optimal enforcement with respect to answer precision, i.e., with the same amount of singletons.

*8.2.2 Performance.* The total running time of Spire for each instance consists of: *(i)* Bayesian inference (done by Psi) and *(ii)* synthesis time (done by either SynSMT or SynGrd). For each instance we measured the running times of Psi and the synthesis algorithms SynSMT and SynGrd.

***Bayesian Inference.*** The running times of Psi are shown in Figure 8. We observe that Psi running times increase with the size of the input set $\mathcal{I}$. Our synthesis algorithms, however, are independent of the inference engine and we can thus directly benefit from any advances in this area. As an interesting item for future work, we plan to experiment with sound probabilistic abstractions, as in [37].

***SynSMT.*** The size of the SMT constraints, stored in the SmtLib2 format, ranged up to 78KB for the medical scenario, up to 263KB for social scenario, and up to 18MB for the location scenario. We plot the performance of SynSMT in Figure 9, where the $X$-axis shows the instance and the $Y$-axis the time in seconds. Each line in the plots shows the times for a particular program.

The plots indicate that SynSMT is time-demanding, since its running time increases roughly exponentially in the size of the set of outputs. This explains the almost constant running times of SynSMT for the "read" programs in the medical and the social scenario, where the output size is constant.

***SynGrd.*** SynGrd always finishes well below one second, hence the synthesis of permissive enforcements is feasible on large instances. We do not explicitly plot SynGrd's performance on our benchmarks; instead, next we present results that demonstrate how SynGrd scales to large synthesis instances.

***Scaling to Large Synthesis Instances.*** To evaluate SynGrd's scalability, we randomly generated large instances. The distribution of outputs $\mathbb{P}^{\pi}_{\delta}(O = o)$ was chosen by first assigning each output an uniformly randomly chosen value from $[0, 1]$ and then normalizing the values to get a distribution. We used a policy with one secret, and a security assertion with bounds $[0.35, 0.65]$. The probabilities of secret for the individual outputs $\mathbb{P}^{\pi}_{\delta}(I \in S \mid O = o)$ ware chosen uniformly randomly from $[0, 1]$.

The running times are shown in Figure 10 in seconds. The figure demonstrates that SynGrd runs in quadratic time in the number of outputs $|O|$.

*8.2.3 Comparison to Conservative Approaches.* To evaluate the general permissiveness of our approach, we compare Spire to the conservative approach of [37], which rejects the attacker's program whenever it does not satisfy the policy. To evaluate this, for each

instance with a program $\pi$, an attacker belief $\delta$, and privacy policy $\Phi$, we checked whether $\pi, \delta \models \Phi$. This indicates whether the conservative approach rejects the attacker's program.

The percentage of rejected programs by the conservative approach is 100% for the medical, 70.84% for the social, and 75% for the location scenario.

*8.2.4 Permissiveness for Interactive Attackers.* To evaluate the iterative mode of Spire, we iterated one representative program from each scenario. In Figure 11a, we measured the relative permissiveness for each iteration and each program. The X-axis indicates the iteration number, and the Y-axis the ratio of equivalence classes of the enforcement to the number of outputs with non-zero probability, i.e. $\frac{|O_i/\xi_i|}{|O_i|}$, where $O_i$ is the set of outputs with non-zero probability in the $i$-th iteration and $\xi_i$ is the enforcement synthesized in the $i$-th iteration. In Figure 11b, we plot how the number of outputs with a non-zero probability reduces as the attacker belief evolves in each consecutive iteration, i.e. $\frac{|O_i|}{|O_1|}$, where $O_i$ is the number of outputs with non-zero probability in the $i$-th iteration. Initially, the program form the medical scenario has 13 outputs in the first iteration, social has 7, and location has 169.

## 9 RELATED WORK

We survey the works that are most closely related to ours.

***Probabilistic Privacy Enforcement.*** Mardizel et al. [37] investigate the problem of enforcing privacy policies that are formalized as thresholds on the attacker belief. In [37], the key contribution is the probabilistic polyhedra abstract domain, which is used to efficiently implement (based on abstract interpretation [15]) sound approximate inference. In contrast to [37], we focus on synthesizing an optimally permissive enforcement that enforces the given privacy policy. The focus of [37] is to efficiently check $\pi, \delta \models \Phi$ for a program $\pi$, an attacker belief $\delta$, and a policy $\Phi$, while our focus is to find an optimally permissive enforcement $\xi$ with $\textsc{Enf}(\pi, \xi), \delta, \models \Phi$. An interesting direction for future work would be to combine our approach with the abstractions of [37].

Guarnieri et al. [27] instantiates Mardziel et al.'s framework [37] to the database setting, where programs consist of relational calculus queries and the attacker belief is formalized using probabilistic logic programs. They develop a provably secure enforcement mechanism that prevents information leakage in the presence of probabilistic dependencies. Their mechanism leverages a dedicated inference engine for a class of probabilistic logic programs to ensure tractability.

Besson et al. [7] randomize the program's inputs (while Spire randomizes over outputs) to enforce privacy in the context of browser fingerprinting by synthesizing a new program using linear programming. The two approaches are non-comparable (even though they both encode bounds on probabilities as linear programs): *(i)* assertions: Spire supports arbitrary assertions over the attacker belief, while [7] considers a specific one (related to browser fingerprinting), *(ii)* reduction: Spire reduces to a linear optimization problem over SMT, while the algorithm of [7] reduces to an LP problem, and *(iii)* Spire offers a full end-to-end implementation while [7] does not offer any implementation or a system.

***Language-based Security.*** Standard non-interference notions have been extended to support probabilities [42, 56] for concurrent programs. Our security notion, however, differs from non-interference in that it allows leaks of sensitive information as long as these do not violate the privacy policy.

Schoepe et al. [48] formalize opacity, a security property that allows any leak except leaking whether a secret holds. Our privacy policies extend opacity to the probabilistic setting. Moreover, since we synthesize the most permissive secure enforcement, a program secured by Spire returns meaningful results even if it is not opaque.

Recently, the language-based security community focused on Quantitative Information Flow [10, 49], where the goal is to quantify the amount of information leaked by a program. Our goal is not to quantify the amount of leaked information. Instead, we synthesize the enforcement that prevents only the information leaks that do not conform to the privacy policy.

***Opacity for Discrete Event Systems.*** There has been a growing interest in opacity for Discrete Event Systems (DES) [32], where systems are usually formalized as Labelled Transition Systems (LTSs) and secrets as predicates over runs. Many flavors of opacity have been studied [12, 43] and recent approaches extend opacity to probabilistic systems [5, 6, 44]. The DES community focused on *(i)* verifying opacity [5, 28, 44], *(ii)* synthesis of mechanisms that enforce opacity [18, 58], and *(iii)* runtime enforcement of opacity [21]. While verification techniques exist for both deterministic [28] and probabilistic systems [5, 44], existing synthesis techniques support only deterministic secrets [18, 21, 58].

***Differential Privacy.*** Differential Privacy [20] has emerged as a standard for protecting statistical databases and privacy-preserving data analysis. A differentially private computation ensures that the presence (or absence) of an individual's data in the input has a little (and bounded) impact on the output. Differential Privacy makes no assumptions on the attacker belief. In contrast, we assume that the attacker belief is known, and we synthesize the most permissive enforcement relation that complies with the privacy policy.

***Probabilistic Programming.*** Numerous probabilistic programming languages have been developed in the past few years: Stan [24], PSI [22], Fabular [11], Anglican [57], Church [25], Venture [36], R2 [38]. These languages support different inference methods (e.g. via translation to Bayesian Nets, sampling, exact). For a detailed survey see [26]. Abstractions for probabilistic programming have been also developed; see [37]. In our work, we leverage existing inference engines to compute the amount of information leaked by the program. The recent developments in probabilistic programming languages are thus orthogonal and beneficial to our approach.

## 10 CONCLUSION

We introduced the problem of optimal privacy enforcement synthesis. The goal is to automatically synthesize an enforcement that transforms a program into a policy-compliant one. We showed that determining the amount of leaked information by the program can be done by probabilistic analysis using existing probabilistic programming engines.

We proved that finding an optimally permissive enforcement is NP-equivalent, and presented an algorithm that reduces the enforcement synthesis problem to a linear optimization problem over SMT

constraints. We also presented a sound greedy synthesis algorithm that runs in quadratic time.

We presented the Spire system, an end-to-end implementation of our synthesis approach. We evaluated Spire on 10 relevant programs from different application domains. Our results demonstrate that permissive privacy enforcement synthesis is feasible and that Spire can handle nontrivial programs. Further, our greedy synthesis performs well on all of our examples: it synthesizes an enforcement in < 1s, the synthesized enforcements are optimal in 89% of the synthesis instances, and overall the enforcements are 95% as permissive as the optimal enforcements.

In the future, we plan to instantiate and optimize the core SPIRE approach for specific application domains (e.g., network privacy), as well as investigate a possible generalization of our permissiveness theorems.

## REFERENCES


[1] 2017. OpenSNP - A Crowdsourced Web Resource for Personal Genomics. https://opensnp.org/snps/rs11200638. (2017).

[2] 2017. Privacy of Genomics. https://www.genome.gov/27561246/privacy-in-genomics/. (2017).

[3] Michael Backes, Pascal Berrang, Mathias Humbert, and Praveen Manoharan. 2016. Membership Privacy in MicroRNA-based Studies. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 319–330. https://doi.org/10.1145/2976749.2978355

[4] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. 2010. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, Vol. 13. 14.

[5] Béatrice Bérard, Krishnendu Chatterjee, and Nathalie Sznajder. 2015. Probabilistic opacity for Markov decision processes. *Inform. Process. Lett.* 115, 1 (2015), 52–59.

[6] Béatrice Bérard, John Mullins, and Mathieu Sassolas. 2015. Quantifying opacity. *Mathematical Structures in Computer Science* 25, 02 (2015), 361–403.

[7] Frédéric Besson, Nataliia Bielova, and Thomas Jensen. 2014. Browser randomisation against fingerprinting: A quantitative information flow approach. In *Nordic Conference on Secure IT Systems*. Springer, 181–196.

[8] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. $\nu$Z-an optimizing SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 194–199.

[9] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos Keromytis. 2017. The KeyNote Trust-Management System. http://www.cs.columbia.edu/~angelos/keynote.html. (2017).

[10] Nicolás E. Bordenabe and Geoffrey Smith. 2016. Correlated Secrets in Quantitative Information Flow. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*. 93–104. https://doi.org/10.1109/CSF.2016.14

[11] Johannes Borgström, Andrew D. Gordon, Long Ouyang, Claudio Russo, Adam Ścibior, and Marcin Szymczak. 2016. Fabular: Regression Formulas As Probabilistic Programming *(POPL)*.

[12] Jeremy W. Bryans, Maciej Koutny, Laurent Mazaré, and Peter Y. A. Ryan. 2008. Opacity generalised to transition systems. *International Journal of Information Security* 7, 6 (2008), 421–435. https://doi.org/10.1007/s10207-008-0058-x

[13] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. 2006. Anonymity protocols as noisy channels. In *International Symposium on Trustworthy Global Computing*. Springer, 281–300.

[14] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. 2009. Quantifying Information Flow with Beliefs. *J. Comput. Secur.* 17, 5 (Oct. 2009), 655–701. http://dl.acm.org/citation.cfm?id=1662658.1662660

[15] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*. ACM, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[17] Andrew DeWan, Mugen Liu, Stephen Hartman, Samuel Shao-Min Zhang, David T. L. Liu, Connie Zhao, Pancy O. S. Tam, Wai Man Chan, Dennis S. C. Lam, Michael Snyder, Colin Barnstable, Chi Pui Pang, and Josephine Hoh. 2006. HTRA1 Promoter Polymorphism in Wet Age-Related Macular Degeneration. 314, 5801 (2006), 989–992. https://doi.org/10.1126/science.1133807

[18] Jérémy Dubreil, Philippe Darondeau, and Hervé Marchand. 2008. Opacity enforcing control synthesis. In *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*. IEEE, 28–35.

[19] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *Proceedings of the Third Conference on Theory of Cryptography (TCC'06)*. Springer-Verlag, Berlin, Heidelberg, 265–284. https://doi.org/10.1007/11681878_14

[20] Cynthia Dwork and Aaron Roth. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science* 9, 3-4 (2014), 211–407.

[21] Ylies Falcone and Hervé Marchand. 2015. Enforcement and validation (at runtime) of various notions of opacity. *Discrete Event Dynamic Systems* 25, 4 (2015), 531–570.

[22] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. *PSI: Exact Symbolic Inference for Probabilistic Programs*. Springer International Publishing, Cham, 62–83. https://doi.org/10.1007/978-3-319-41528-4_4

[23] Johannes Gehrke, Edward Lui, and Rafael Pass. 2011. Towards Privacy for Social Networks: A Zero-knowledge Based Definition of Privacy. In *Proceedings of the 8th Conference on Theory of Cryptography (TCC'11)*. Springer-Verlag, Berlin, Heidelberg, 432–449. http://dl.acm.org/citation.cfm?id=1987260.1987294

[24] Andrew Gelman, Daniel Lee, and Jiqiang Guo. 2015. Stan: A Probabilistic Programming Language for Bayesian Inference and Optimization. *J. Educational and Behavioral Stats.* (2015).

[25] ND Goodman, VK Mansinghka, D Roy, K Bonawitz, and JB Tenenbaum. 2008. Church: A language for generative models. In *UAI*.

[26] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming *(FOSE 2014)*.

[27] Marco Guarnieri, Srdjan Marinovic, and David Basin. 2017. Securing Databases from Probabilistic Inference. *arXiv preprint arXiv:1706.02473* (2017).

[28] Christoforos N Hadjicostis and Christoforos Keroglou. 2014. Opacity formulations and verification in discrete event systems. In *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*. IEEE, 1–12.

[29] Jianming He, Wesley W. Chu, and Zhenyu (Victor) Liu. 2006. Inferring Privacy Information from Social Networks. In *Proceedings of the 4th IEEE International Conference on Intelligence and Security Informatics (ISI'06)*. Springer-Verlag, Berlin, Heidelberg, 154–165. https://doi.org/10.1007/11760146_14

[30] Nils Homer, Szabolcs Szelinger, Margot Redman, David Duggan, Waibhav Tembe, Jill Muehling, John V. Pearson, Dietrich A. Stephan, Stanley F. Nelson, and David W. Craig. 2008. Resolving Individuals Contributing Trace Amounts of DNA to Highly Complex Mixtures Using High-Density SNP Genotyping Microarrays. *PLOS Genetics* 4, 8 (08 2008), 1–9. https://doi.org/10.1371/journal.pgen.1000167

[31] Mathias Humbert. 2015. *When Others Impinge upon Your Privacy: Interdependent Risks and Protection in a Connected World*. Ph.D. Dissertation. École Polytechnique FéDérale de Lanne.

[32] Romain Jacob, Jean-Jacques Lesage, and Jean-Marc Faure. 2015. Opacity of discrete event systems: models, validation and quantification. *IFAC-PapersOnLine* 48, 7 (2015), 174–181.

[33] Richard M Karp. 1972. Reducibility among combinatorial problems. In *Complexity of computer computations*. Springer, 85–103.

[34] John Krumm. 2009. A Survey of Computational Location Privacy. *Personal Ubiquitous Comput.* 13, 6 (Aug. 2009), 391–399. https://doi.org/10.1007/s00779-008-0212-5

[35] Jack Lindamood, Raymond Heatherly, Murat Kantarcioglu, and Bhavani Thuraisingham. 2009. Inferring Private Information Using Social Network Data. In *Proceedings of the 18th International Conference on World Wide Web (WWW '09)*. ACM, New York, NY, USA, 1145–1146. https://doi.org/10.1145/1526709.1526899

[36] Vikash Mansinghka, Daniel Selsam, and Yura Perov. 2014. Venture: a higher-order probabilistic programming platform with programmable inference. *ArXiv 1404.0099* (2014).

[37] Piotr Mardziel, Stephen Magill, Michael Hicks, and Mudhakar Srivatsa. 2013. Dynamic Enforcement of Knowledge-based Security Policies Using Probabilistic Abstract Interpretation. *J. Comput. Secur.* 21, 4 (July 2013), 463–532. http://dl.acm.org/citation.cfm?id=2590624.2590625

[38] Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. 2014. R2: An efficient MCMC sampler for probabilistic programs. In *AAAI*.

[39] A. M. Olteanu, K. Huguenin, R. Shokri, M. Humbert, and J. P. Hubaux. 2016. Quantifying Interdependent Privacy Risks with Location Data. *IEEE Transactions on Mobile Computing* PP, 99, 1–1.

[40] A. Pfeffer. 2009. *Figaro: An object-oriented probabilistic programming language*. Technical Report. Charles River Analytics.

[41] Andrea J. Richardson, F. M. Amirul Islam, Khin Zaw Aung, Robyn H. Guymer, and Paul N. Baird. 2010. An Intergenic Region between the tagSNP rs3793917 and rs11200638 in the HTRA1 Gene Indicates Association with Age-Related Macular Degeneration. *Investigative Ophthalmology & Visual Science* 51, 10 (2010), 4932. https://doi.org/10.1167/iovs.09-5114 arXiv:/data/journals/iovs/932962/z7g01010004932.pdf

[42] Andrei Sabelfeld and David Sands. 2000. Probabilistic Noninterference for Multi-Threaded Programs. In *Proceedings of the 13th IEEE Workshop on Computer Security Foundations (CSFW '00)*. IEEE Computer Society, Washington, DC, USA, 200–. http://dl.acm.org/citation.cfm?id=794200.795151

[43] Anooshiravan Saboori and Christoforos N Hadjicostis. 2007. Notions of security and opacity in discrete event systems. In *Decision and Control, 2007 46th IEEE Conference on*. IEEE, 5056–5061.
[44] Anooshiravan Saboori and Christoforos N Hadjicostis. 2014. Current-state opacity formulations in probabilistic finite automata. *IEEE Transactions on automatic control* 59, 1 (2014), 120–133.
[45] Pierangela Samarati. 2001. Protecting Respondents' Identities in Microdata Release. *IEEE Transactions on Knowledge and Data Engineering* 13, 6 (2001), 1010–1027.
[46] Pierangela Samarati and Latanya Sweeney. 1998. *Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression.* Technical Report. Technical report, SRI International.
[47] Taisuke Sato. 2008. A Glimpse of Symbolic-statistical Modeling by PRISM. *J. Intell. Inf. Syst.* 31, 2 (Oct. 2008), 161–176. https://doi.org/10.1007/s10844-008-0062-7
[48] Daniel Schoepe and Andrei Sabelfeld. 2015. Understanding and Enforcing Opacity. In *2015 IEEE 28th Computer Security Foundations Symposium*. IEEE, 539–553.
[49] Geoffrey Smith. 2009. On the foundations of quantitative information flow. In *International Conference on Foundations of Software Science and Computational Structures*. Springer, 288–302.
[50] David Sommer, Aritra Dhar, Luka Malisa, Esfandiar Mohammadi, Daniel Ronzani, and Srdjan Capkun. 2017. CoverUp: Privacy Through "Forced" Participation in Anonymous Communication Networks. In *ASIA CCS*. ACM. https://doi.org/10.1145/3052973.3056126
[51] Latanya Sweeney. 2002. Achieving k-anonymity privacy protection using generalization and suppression. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10, 05 (2002), 571–588.
[52] Manolis Terrovitis. 2011. Privacy Preservation in the Dissemination of Location Data. *SIGKDD Explor. Newsl.* 13, 1 (Aug. 2011), 6–18. https://doi.org/10.1145/2031331.2031334
[53] Petar Tsankov, Mohammad Torabi Marinovic, and David Basin. 2016. Access Control Synthesis for Physical Spaces. In *CSF*.
[54] Petar Tsankov, Srdjan Marinovic, Mohammad Torabi Dashti, and David Basin. 2014. Decentralized Composite Access Control. In *POST*.
[55] L.G. Valiant. 1979. The complexity of computing the permanent. *Theoretical Computer Science* 8, 2 (1979), 189 – 201. https://doi.org/10.1016/0304-3975(79)90044-6
[56] Dennis Volpano and Geoffrey Smith. 1999. Probabilistic noninterference in a concurrent language1. *Journal of Computer Security* 7, 2-3 (1999), 231–253.
[57] Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. 2014. A New Approach to Probabilistic Programming Inference. In *AISTATS*.
[58] Yi-Chin Wu and StÃľphane Lafortune. 2014. Synthesis of insertion functions for enforcement of opacity security properties. *Automatica* 50, 5 (2014), 1336 – 1348. https://doi.org/10.1016/j.automatica.2014.02.038
[59] Elena Zheleva and Lise Getoor. 2009. To Join or Not to Join: The Illusion of Privacy in Social Networks with Mixed Public and Private User Profiles. In *Proceedings of the 18th International Conference on World Wide Web (WWW '09)*. ACM, New York, NY, USA, 531–540. https://doi.org/10.1145/1526709.1526781

# A PROOFS OF THEOREMS

In this appendix, we give proofs for the theorems.

## A.1 Complexity

To prove Theorem 4.2, we show that the permissive privacy enforcement synthesis problem (i.e. the optimal enforcement problem with permissiveness chosen as optimality) is both NP-easy and NP-hard.

Lemma A.1. *The permissive privacy enforcement synthesis problem is NP-easy.*

Proof. To prove that the problem is in $\mathsf{FP}^{\mathsf{NP}}$, we first define a decision version of the permissive policy enforcement synthesis problem that we will then use as an oracle.

An instance of the decision problem is a tuple $(\mathcal{I}, \mathcal{O}, \delta, \pi, \Phi, N, \alpha)$ where $\mathcal{I}$ is the input set, $\mathcal{O}$ is the output set, $\pi$ is the probabilistic program over these sets, $\Phi = \{\varphi_1, \dots, \varphi_\ell\}$ is a privacy policy where $\varphi_i = (S_i, [a_i, b_i])$, $N \in \mathbb{N}$, and $\alpha \subseteq \mathcal{O} \times \mathcal{O}$. The problem then asks whether there is an equivalence relation $\xi$ over $\mathcal{O}$ such that $\textsc{Enf}(\pi, \xi), \delta \models \Phi$, $|\xi| = N$, and $\alpha \subseteq \xi$.

This decision problem is in NP since a non-deterministic Turing machine can non-deterministically guess a relation $R$ over the set $\mathcal{O}$, and then check whether $R$ is an equivalence relation, $R$ enforces the policy $\Phi$, $|R| = N$, $\alpha \subseteq R$, and accept only if all of these conditions are met, which can be checked in polynomial time.

We now show that the functional problem is in $\mathsf{FP}^{\mathsf{NP}}$ by providing a polynomial algorithm with an oracle for the decision problem:

1. We determine the maximum possible size of an equivalence relation enforcing $\Phi$. We use binary search on the interval $[0, |\mathcal{O}|]$. To determine if an equivalence relation of size $k$ exists, we query the oracle with an instance $(\mathcal{I}, \mathcal{O}, \delta, \pi, \Phi, k, \emptyset)$. Let the maximum size be $K$.
2. We find an equivalence relation of size $K$ enforcing the policy $\Phi$. To do this, we initialize $\xi \leftarrow \emptyset$ and $\hat{\xi} \leftarrow \mathcal{O} \times \mathcal{O}$ and iterate the following procedure until $\hat{\xi} \neq \emptyset$:
   (a) Pick an arbitrary element $(o, o')$ from $\hat{\xi}$. Query the oracle with $(\mathcal{I}, \mathcal{O}, \delta, \pi, \Phi, K, \xi \cup \{(o, o')\})$
   (b) If the answer is Yes, then $\xi \leftarrow \xi \cup \{(o, o')\}$. If the answer is No, do nothing.
   (c) $\hat{\xi} \leftarrow \hat{\xi} \setminus \{(o, o')\}$
   (d) Iterate until $\hat{\xi} \neq \emptyset$.

   After the iteration ends, we just output $\xi$.

□

Lemma A.2. *The permissive privacy enforcement synthesis problem is* NP*-hard.*

Proof. To prove the NP-hardness, we define a refined version of the problem, then we show the refined problem is NP-hard by reducing the partition problem to it. Finally, we reduce the refined problem to the permissive privacy enforcement synthesis problem and hence prove it to be NP-hard as well.

An instance of the refined problem is a set $O$, its subset $S$, a distribution $d$ over $O$ ($d \in \mathcal{D}(O)$) and a bound $[a, b]$ where $a, b \in \mathbb{Q}$ and $0 \le a \le b \le 1$. The problem asks to find an equivalence relation $\xi$ over $O$ such that for each class $E \in \xi$ we have that:

$$\Pr(O \in S \mid O \in E) = \frac{\sum_{o \in E \cap S} d(o)}{\sum_{o \in E} d(o)} \in [a, b]$$

The refined problem is trivially reducible to the permissive policy enforcement problem. For an instance $(O, S, d, [a, b])$ of the refined problem, we produce an instance of the permissive policy enforcement problem:

$$(\mathcal{I} = O, \mathcal{O} = O, \delta = d, \pi = I_O, \Phi = \{\varphi = (S, [a, b])\})$$

where $I_O \in \mathbb{M}_{O \times O}$ is the identity matrix.

Let us reduce the partition problem, which is known to be NP-hard [33], to the refined problem. The partition problem is defined as follows. Let $A = \{a_1, \dots, a_n\}$ be a finite set of natural numbers. The partition problem asks to decide whether there exists a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} a = \sum_{a \in A \setminus A'} a$$

We now construct an instance of the refined problem: we choose the set $O = \{o_1, \dots, o_n, p, q\}$, and $S = \{p, q\}$. We choose the prior distribution as:

$$d(p) = d(q) = 0.25$$

$$d(o_i) = \frac{a_i}{2 \cdot \sum_i a_i} \text{ where } a_i \in A$$

And we choose the bound $[0.5, 0.5]$.

With this instance, it is possible to find the desired subset $A'$ of $A$ if and only if there exists an equivalence class over outputs with exactly 2 equivalence classes. When $\xi = \{O_1, O_2\}$ is an equivalence relation solving the refined problem, we have that $p \in O_1 \wedge q \in O_2$, or $p \in O_2 \wedge q \in O_1$. We can get the solution to the partition problem as $A' = \{a_i \mid o_i \in O_1\}$. □

## A.2 Completeness

Proof of Theorem 4.4. Let $(S, [a, b]) \in \Phi$ be a belief bound. We show that $\text{Enf}(\pi', \xi_\perp) \models (S, [a, b])$. We have that $\pi, \delta \models (S, [a, b])$, which is by definition $\mathbb{P}^\pi_\delta(I \in S \mid O = o) \in [a, b]$ for every $o \in \mathcal{O}$. The collection of events $\{O = o\}_{o \in \mathcal{O}}$ is a partition of the sample space. By the law of total probability, we have:

$$\mathbb{P}^\pi_\delta(I \in S) = \sum_{o \in \mathcal{O}} \mathbb{P}^\pi_\delta(I \in S \mid O = o) \cdot \mathbb{P}^\pi_\delta(O = o)$$

This gives us that $\mathbb{P}^\pi_\delta(I \in S)$ is a convex combination of numbers in the range $[a, b]$ since $\sum_{o \in \mathcal{O}} \mathbb{P}^\pi_\delta(O = o) = 1$. Hence, $\mathbb{P}^\pi_\delta(I \in S) \in [a, b]$. Finally,

$$\mathbb{P}^\pi_\delta(I \in S) = \sum_{i \in S} \delta(i) = \mathbb{P}^{\pi'}_\delta(I \in S \mid O \in \mathcal{O})$$

since $\{O \in \mathcal{O}\} = \Omega$, which proves the theorem. □

## A.3 Algorithms

Proof of Theorem 5.2. $\ker(\mathcal{M})$ is an equivalence relation by definition. The assertion of $\psi_{\text{bounds}}$ is equivalent to $\ker(\mathcal{M})$ enforcing $\Phi$ and maximizing $\psi_{\text{obj}}$ is equivalent to $\ker(\mathcal{M})$ being optimal. □

Proof of Theorem 6.2. Suppose there is no enforcement $\xi$ such that $\text{Enf}(\pi, \xi), \delta \models \Phi$. Then, there must be a belief bound $(S, [a, b]) \in \Phi$ such that $\mathbb{P}^\pi_\delta(I \in S) \notin [a, b]$. Otherwise, we would have that $\text{Enf}(\pi, \xi_\perp), \delta \models \Phi$. Hence, SynGrd returns unsat.

Now, suppose there is an equivalence relation enforcing the policy. The while loop on line 5 joins two classes of the equivalence relation $\xi$ (declared on Line 5) in every iteration. After at most $|O|$ iterations, we have $\xi = \xi_\perp$, for which the while condition on Line 9 must be satisfied. Also, the while condition corresponds to $\text{Enf}(\pi, \xi), \delta \not\models \Phi$, so when the loop terminates, $\xi$ satisfies the policy. Hence, SynGrd terminates and returns an equivalence relation enforcing the policy. □

## A.4 Optimal Algorithm for Answer Precision with Singleton Policies

Here, we present an algorithm that produces an optimal enforcement with regard to answer precision for singleton policies (we only enforce bounds on the probability of one secret).

The main idea of Algorithm 3 is to join all the outputs that violate the policy into one class $C$, since these can never be singletons in a correct enforcement. Then, if $C$ satisfies the policy, we are done. If not, we must add additional elements to the new class $C$. To pick them correctly, we consider the following. To ensure that $\mathbb{P}^\pi_\delta(I \in S \mid O \in C) \in [a, b]$ means that:

$$\frac{\sum_{o \in C} \mathbb{P}^\pi_\delta(I \in S \mid O = o) \cdot \mathbb{P}^\pi_\delta(O = o)}{\sum_{o \in C} \mathbb{P}^\pi_\delta(O = o)} \in [a, b]$$

Now, $\mathbb{P}^\pi_\delta(I \in S \mid O \in C) \in [a, b]$ can be violated on at most one side (i.e. it can be less that $a$ or greater than $b$, but not both). Since the cases are symmetrical, let us assume without a loss of generality that $\mathbb{P}^\pi_\delta(I \in S \mid O \in C) > b$. Hence we need to ensure that:

$$\frac{\sum_{o \in C} \mathbb{P}^\pi_\delta(I \in S \mid O = o) \cdot \mathbb{P}^\pi_\delta(O = o)}{\sum_{o \in C} \mathbb{P}^\pi_\delta(O = o)} \leq b$$

which is equivalent to

$$\sum_{o \in C} (\mathbb{P}^\pi_\delta(I \in S \mid O = o) - b) \cdot \mathbb{P}^\pi_\delta(O = o) \leq 0$$

Hence, we can keep adding elements from $O \setminus C$ that minimize the expression $(\mathbb{P}^\pi_\delta(I \in S \mid O = o) - b) \cdot \mathbb{P}^\pi_\delta(O = o)$ until $C$ satisfies the belief bound. In the case that there is no enforcement satisfying the policy, we find out after joining all the elements together.

**Algorithm 3:** The algorithm $\text{SynOpt}(\pi, \delta, \Phi)$

```
Input: A probabilistic program π, an attacker belief δ, and a
       policy Φ = {(S, [a, b])}
Output: An equivalence relation ξ enforcing the policy.
1  begin
2    C ← {o ∈ O | P^π_δ(I ∈ S | O = o) ∉ [a, b]}
3    X ← O \ C
4    while P^π_δ(I ∈ S | O ∈ C) ∉ [a, b] ∧ X ≠ ∅ do
5      if P^π_δ(I ∈ S | O ∈ C) < a then
6        o ← arg max_{o∈X} (P^π_δ(I ∈ S | O =
           o) − a) · P^π_δ(O = o)
7      else
8        o ← arg min_{o∈X} (P^π_δ(I ∈ S | O = o) − b) · P^π_δ(O =
           o)
9      C ← C ∪ {o}
10     X ← X \ {o}
11   if P^π_δ(I ∈ S | O ∈ C) ∉ [a, b] then
12     return unsat
13   return {C} ∪ {{o} | o ∈ X}
```

***Running Time.*** The running time of Algorithm 3 is $\mathcal{O}(n \log(n))$ where $n = |O|$. This is because the elements of $O$ can be sorted by the value $(\mathbb{P}^\pi_\delta(I \in S \mid O = o) - b) \cdot \mathbb{P}^\pi_\delta(O = o)$, and then picked one by one in an increasing order.

***Correctness.*** We now prove the following theorem.

Theorem A.3. *Let $\pi$ be a probabilistic program, $\delta$ an attacker belief, and $\Phi = (S, [a, b])$ a singleton privacy policy. If there is no equivalence relation $\xi$ such that $\text{Enf}(\pi, \xi), \delta \models \Phi$, then $\text{SynOpt}(\pi, \delta, \Phi) =$ unsat. Otherwise, we have $\text{SynOpt}(\pi, \delta, \Phi) = \xi$, $\text{Enf}(\pi, \xi), \delta \models \Phi$ and furthermore, we have that $\xi$ has the greatest number of singleton classes of all enforcements enforcing the policy $\Phi$ on $\pi$.*

```
1  def inherit(mother: R[], father: R[]): R[] {
2    return [mother[flip(1/2)], father[flip(1/2)]];
3  }
4  def prior() {
5    nucl := array(7, array(2, 0));
6    nucl[0] := [flip(1/2), flip(1/2)];
7    nucl[1] := [flip(1/2), flip(1/2)];
8    nucl[2] := [flip(1/2), flip(1/2)];
9    nucl[3] := [flip(1/2), flip(1/2)];
10   nucl[4] := inherit[nucl[0], nucl[1]];
11   nucl[5] := inherit[nucl[2], nucl[3]];
12   nucl[6] := inherit[nucl[4], nucl[5]];
13   return nucl;
14 }
```

**(a) medical-belief.psi**

```
1  def program(nucl: R[][]): R {
2    sum := 0;
3    for i := 0 .. 6 {
4      sum := nucl[i][0] + nucl[i][1];
5    }
6    return sum;
7  }
```

**(b) medical-sum.psi**

```
1  Pr[nucl[0][0] == 1 && nucl[0][1] == 1] in [0.1, 0.9]
```

**(c) medical-policy.txt**

**Figure 12: (a) Attacker belief, (b) sum program, and (c) privacy policy for the medical scenario**

Proof of Theorem A.3. When no enforcement exists, the loop on Line 4 will iterate until all the outputs are conflated. Then, the condition on Line 11 will evaluate to true and an unsat result will be returned.

For the case that an enforcement exists, let $\xi^*$ be an optimal enforcement for the program $\pi$, belief $\delta$ and policy $\Phi$. Without a loss of generality we can assume that $\xi^*$ has only singleton classes with the exception of a single class $C^*$ where all the other outputs are located. Now, if SynOpt would give an enforcement with less singletons than $\xi^*$ has, it would be a contradiction, since it must hold that

$$\sum_{o \in C^*} (\mathbb{P}^{\pi}_{\delta}(I \in S \mid O = o) - b) \cdot \mathbb{P}^{\pi}_{\delta}(O = o) \leq 0$$

and SynOpt initializes $C$ to be the subset of $C^*$ that violates the policy and then picks the outputs to conjoin to $C$ exactly by $(\mathbb{P}^{\pi}_{\delta}(I \in S \mid O = o) - b) \cdot \mathbb{P}^{\pi}_{\delta}(O = o)$. □

Since the running time of Algorithm 3 is polynomial, by Theorem A.3, we conclude that the synthesis problem for singleton policies is in PTIME for answer precision optimality. This immediately proves Theorem 4.3.

# B PSI SYNTAX

We present the syntax of the Psi solver language in BNF:

$$x \in \mathit{Vars} \quad a \in \mathit{ArrayVars} \quad \mathit{bop} \in \{+, -, *, /, ==, \neq, <, \leq\}$$

$$\mathit{Expr} ::= \mathbb{Q} \mid x \mid a[\mathit{Expr}] \mid \mathit{Expr}\ \mathit{bop}\ \mathit{Expr} \mid \texttt{flip}(\mathit{Expr})$$

$$\mathit{Stmt} ::= x := \mathit{Expr} \mid a := \texttt{array}(\mathit{Expr}) \mid x = \mathit{Expr} \mid a[\mathit{Expr}] = \mathit{Expr}$$
$$\mid \texttt{skip} \mid \texttt{observe}\ \mathit{Expr} \mid \texttt{if}\ \mathit{Expr}\ \{\mathit{Stmt}\}\ \texttt{else}\ \{\mathit{Stmt}\}$$
$$\mid \texttt{for}\ x\ \texttt{in}\ [\mathit{Expr}..\mathit{Expr})\ \{\mathit{Stmt}\} \mid \mathit{Stmt};\ \mathit{Stmt}$$

# C EXPERIMENT DETAILS

We generated synthesis instances from three scenarios.

## C.1 Genomic Data

The genomic data scenario is identical to the one described in Section 2. We generated synthesis instances with different number of patients and privacy policies. We denote the instance with $n$ patients and $m$ privacy policies by $(n, m)$.

***Attacker Belief.*** For an instance $(n, m)$, the set of inputs is $\mathcal{I} = \{\texttt{A}, \texttt{G}\}^{2n}$. The relationships among patients form a complete binary tree with $n$ nodes. The belief $\delta \in \mathcal{D}(\mathcal{I})$ is defined as described in Figure 2(b).

***Policy.*** For an instance $(n, m)$, we generate a privacy policy with $m$ belief bounds of the form $(I[i] = \texttt{AA}, [0.1, 0.9])$, where $I[i]$ returns the pair of nucleotides of patient $i$ according to the topological order.

***Programs.*** We consider four programs:

- `sum` Returns the number of adenine nucleotides among the patients.
- `noisy-sum` Returns the number of adenine nucleotides among the patients with noise, which adds 1 with probability 0.5 and subtracts 1 with probability 0.5.
- `prevalence` Returns the number of patients who have two adenine nucleotides.
- `read` Returns the pair of nucleotides of a patient.

## C.2 Social

We model a social network where users express whether they favor the Democratic or the Republican party. As before, we denote by $(n, m)$ a instance with $n$ users and $m$ policies.

***Attacker Belief.*** For an instance $(n, m)$, the set of inputs is $\mathcal{I} = \{\texttt{D}, \texttt{R}\}^{n}$. We add a friendship between two users randomly with a probability 0.5. The attacker belief is defined by a probabilistic program that assigns a Bernoulli distribution to the affiliation to each user $i$ with the statement `affi[i] := flip(0.5)`, and then for any friendship between users $i$ and $j$ we add an observe statement

```
1  if (flip(0.5)) { observe(affi[i] == affi[j]); }
```

to correlate their affiliations.

***Policy.*** For an instance $(n, m)$ we generate a privacy policy with $m$ belief bounds of the form (`affi[i] == R`, $[0.1, 0.9]$), where $i$ ranges over $m$ randomly selected users. A belief bound for a user imposes a limit on how much the attacker can learn about the user's affiliation.

***Programs.*** We consider three programs:

- `sum` Returns the number of Republicans among the users.
- `noisy-sum` Returns the number of of Republicans among the users with noise, which adds 1 with probability 0.5 and subtracts 1 with probability 0.5.
- `read` Returns the affiliation of a user.

```
1  def prior() {
2    affiliation := array(N, flip(1/2));
3    friends := [[0, 1, ..., 0],
4                [1, 0, ..., 0],
5                 .  .       .
6                 .  .       .
7                 .  .       .
8                [0, 0, ..., 0]];
9    for i := 0 .. N-1 {
10     for j := i+1 .. N-1 {
11       if(friends[i][j] && flip(1/2)) {
12         observe(affiliation[i] == affiliation[j]);
13       }
14     }
15   }
16   return affiliation;
17 }
```

**(a) social-attacker-belief.psi**

```
1  def program(affiliation: R[]) {
2    sum := 0;
3    for i := 0 .. N-1 {
4      sum += affiliation[i];
5    }
6    return sum;
7  }
```

**(b) social-sum.psi**

```
1  Pr[affiliation[0] == 1] in [0.1, 0.9]
```

**(c) social-policy.txt**

**Figure 13: (a) Attacker belief, (b) sum program, and (c) privacy policy for the social scenario**

```
1  def prior() {
2    x := uniformInt(0,15);
3    y := uniformInt(0,15);
4    return (x, y);
5  }
```

**(a) location-attacker-belief.psi**

```
1  def program(x, y) {
2    return (x, y);
3  }
```

**(b) location-identity.psi**

```
1  Pr[(x >= 5) && (x <= 7) && (y >= 12) && (y <= 14)] in [0, 0.5]
```

**(c) social-policy.txt**

**Figure 14: The programs for the location scenario with the identity program.**

## C.3 Location

In this scenario we consider location-based services that query the user's location [39]. We generate synthesis instances $(n, m)$ with area of size $n \times n$ and $m$ protected areas of size $3 \times 3$, which are placed at random positions.

***Attacker Belief.*** For an instance $(n, m)$, the set of inputs is $\mathcal{I} = \{1, \ldots, n\} \times \{1, \ldots, n\}$. All positions in the grid are equally likely.

***Policy.*** For an instance $(n, m)$ we generate a privacy policy with $m$ bounds of the form $(I \in A, [0.1, 0.9])$ where $A$ is the set of positions that define a protected area.

***Program.*** We consider three programs:

- read: Returns the user's location directly.
- constant: If the user is in a protected area, returns $(0, 0)$, otherwise returns the user's location.
- random: If the user is in a protected area, returns a random location, otherwise the user's location.