Verified Foundations for Differential Privacy

MARKUS DE MEDEIROS, New York University, United States
MUHAMMAD NAVEED, Amazon, United States
TANCREDE LEPOINT, Amazon, United States
TEMESGHEN KAHSAI, Amazon, United States
TRISTAN RAVITCH, Amazon, United States
STEFAN ZETZSCHE, Amazon, United Kingdom
ANJALI JOSHI, Amazon, United States
JOSEPH TASSAROTTI, New York University, United States
AWS ALBARGHOUTHI, Amazon, United States
JEAN-BAPTISTE TRISTAN*, Amazon, United States

Differential privacy (DP) has become the gold standard for privacy-preserving data analysis, but implementing it correctly has proven challenging. Prior work has focused on verifying DP at a high level, assuming the foundations are correct and a perfect source of randomness is available. However, the underlying theory of differential privacy can be very complex and subtle. Flaws in basic mechanisms and random number generation have been a critical source of vulnerabilities in real-world DP systems.

In this paper, we present SampCert, the first comprehensive, mechanized foundation for differential privacy. SampCert is written in Lean with over 12,000 lines of proof. It offers a generic and extensible notion of DP, a framework for constructing and composing DP mechanisms, and formally verified implementations of Laplace and Gaussian sampling algorithms. SampCert provides (1) a mechanized foundation for developing the next generation of differentially private algorithms, and (2) mechanically verified primitives that can be deployed in production systems. Indeed, SampCert's verified algorithms power the DP offerings of Amazon Web Services (AWS), demonstrating its real-world impact.

SampCert's key innovations include: (1) A generic DP foundation that can be instantiated for various DP definitions (e.g., pure, concentrated, Rényi DP); (2) formally verified discrete Laplace and Gaussian sampling algorithms that avoid the pitfalls of floating-point implementations; and (3) a simple probability monad and novel proof techniques that streamline the formalization. To enable proving complex correctness properties of DP and random number generation, SampCert makes heavy use of Lean's extensive Mathlib library, leveraging theorems in Fourier analysis, measure and probability theory, number theory, and topology.

1 INTRODUCTION

Differential privacy (DP) [19] has become the gold standard for privacy-preserving data analysis. It has been widely adopted in industry [4, 15], government [1], and academia as a robust framework for publishing sensitive data while providing strong privacy guarantees. Despite its ubiquity, implementing differential privacy correctly has proven to be a significant challenge. Bugs have been, and continue to be, discovered in the random number generation algorithms [33] and differentially private mechanisms used in real-world systems [29, 31, 41].

Prior work has explored the verification and testing of DP at a high level of abstraction [2, 3, 7, 8, 10, 16, 22, 24, 25, 32, 37–39, 42, 44, 45, 47, 48], assuming that the foundations of differential privacy are correct and that a perfect source of randomness is available. This approach is unsustainable given the increasing complexity of modern definitions of differential privacy, and there is a history of critical DP vulnerabilities originating from subtle flaws in random number generation [33].

In this paper, we present SampCert[1], the first comprehensive, mechanized foundation for differential privacy. SampCert is written in the Lean v4 [36] language and theorem prover and uses more than 12,000 lines of proof. SampCert offers a generic and extensible notion of differential privacy, a

---

*Corresponding author, trjohnb@amazon.com
[1]https://github.com/leanprover/SampCert

framework for constructing and composing differentially private mechanisms, and mechanically verified implementations of the Laplace and Gaussian sampling algorithms.

We see SampCert being used in two ways: (1) The mechanized DP foundation provides researchers with a powerful starting point for developing and proving the correctness of new privacy mechanisms and definitions. (2) The mechanically verified primitives, like the random sampling algorithms, can be extracted from Lean and directly deployed to increase assurance of differentially private systems. Indeed, SampCert is used in the AWS Clean Rooms Differential Privacy service.

1.1 Challenges of Verified DP

Working with the theory of DP and building practical realizations of it encounters many challenges:

Challenge 1: The Many Faces of DP. The core challenge in working with differential privacy is deciding on the notion of privacy to use. In addition to the standard definition of *pure* differential privacy [18], there are a plethora of relaxations of this definition that enable the development of more accurate mechanisms while still providing strong guarantees—*approximate* DP [17], *Rényi* DP [34], *zero-concentrated* DP [11], etc. Thus, proofs of mechanisms, as well as practical implementations, tend to fix a definition of DP and assume basic properties about it, e.g., that it composes additively.

Challenge 2: Developing Correct Mechanisms. Computing interesting statistics (means, histograms, gradients, etc.) in a differentially private manner requires developing mechanisms that carefully add random noise at certain points in the computation, and proving their correctness with respect to a definition of privacy. Proving that a mechanism adds the correct type and amount of noise to establish a statistical privacy result turns out to be a difficult task. Indeed, mistakes have been found in commonly used mechanisms like the Sparse Vector Technique [31].

Challenge 3: Sampling Correctly. Finally, for a correct implementation of DP, one requires a sampling algorithm that is faithful to the actual mathematical definition of the probability distribution, which is usually the Laplace or Gaussian distribution. This challenge has been a persistent issue for DP libraries [33]: approximating real-valued sampling algorithms using floating point numbers has led to critical breaches of privacy. To avoid this problem, researchers have proposed theories of discrete DP, which use sampling algorithms that operate over the integers exactly [6, 13].

SampCert addresses each of these issues in turn: we develop an abstract query language that supports generic DP reasoning, a verified mathematical theory for DP (Section 2), and a suite of verified random sampling algorithms which can be efficiently executed (Section 3 and Section 4). SampCert can serve as a foundation to easily develop and prove the correctness of sophisticated DP mechanisms, such as the Sparse Vector Technique.

1.2 Key Ideas

SampCert is designed around several key insights, which put together enable foundational verification of differentially private programs. Fig. 1 shows the high-level architecture of SampCert. Below we describe its key ideas.

A Generic DP Foundation. While the many definitions for DP offer a variety of subtly different statistical guarantees, they also share a set of common properties. For example, the common definitions of differential privacy include *composition theorems*, which specify upper bounds on the privacy of programs built out of private components. In SampCert, we collect a suite of these common idioms into an *abstract* interface for DP. This interface specifies the basic privacy axioms that an instance of DP must satisfy in order to be subject to a generic analysis. Instead of constraining ourselves to one definition of DP—e.g., pure DP or Rényi DP—we can develop our mechanisms
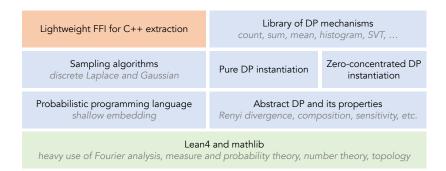
Lightweight FFI for C++ extraction	Library of DP mechanisms *count, sum, mean, histogram, SVT, …*	
Sampling algorithms *discrete Laplace and Gaussian*	Pure DP instantiation	Zero-concentrated DP instantiation
Probabilistic programming language *shallow embedding*	Abstract DP and its properties *Renyi divergence, composition, sensitivity, etc.*	
Lean4 and mathlib *heavy use of Fourier analysis, measure and probability theory, number theory, topology*		

Fig. 1. Overview of SampCert's main components, ordered bottom up by dependency

with respect to the axiomatic definition in a manner parameteric to the underlying definition of DP. In our development we demonstrate how to instantiate the interface for pure differential privacy and zero-concentrated differential privacy, two of the most widely used definitions of DP.

Random Number Generators (The Heart of DP). Differential privacy relies on the ability to sample numbers from, typically, the Laplace or Gaussian distributions. Any mistake in the sampling algorithm destroys the guarantees of differential privacy. Unfortunately it is challenging to sample from these distributions correctly. This is particularly true when dealing with floating-point implementations as poignantly demonstrated by Mironov [33]. Researchers since then have exploited floating-point numbers to mount a series of different attacks [29].

To avoid using floating-point numbers, researchers devised clever algorithms for efficiently sampling from the discrete analogues of the Laplace and Gaussian distributions using exact rational-valued calculations. In this paper, we present a formally verified implementation of the sampling algorithms presented by Canonne et al. [13], which are widely used, for instance, by the US Census Bureau for their DP disclosure of statistics. Specifically, we use Lean to prove that our implementation of the algorithms generates samples following the probability density functions of the discrete Laplace and Gaussian distributions.

A Verification-Ready Probabilistic Programming Language. A key technical challenge with SampCert is how to cleanly specify the sampling algorithms, DP mechanisms, and theorems, while minimizing the proof burden. For specifying probabilistic algorithms, we define a simple, functional probabilistic programming language that manipulates mass functions (which need not be probability mass functions). This simple language is represented as a shallowly-embedded monadic DSL in Lean. By using mass functions rather than full probability spaces (e.g., Hurd's and Giry's monads [23, 27]), we give ourselves a new proof strategy for analyzing looping programs loosely inspired by *liveness* and *safety* properties. In particular, our proof strategy allows us to describe the mass distribution of a loop mathematically (as the pointwise limit of the mass functions of its iterates) without requiring that we construct probabilistic loop invariants that may involve complicated normalizing factors.

Extraction and Deployment. Our simple probabilistic language enables straightforward extraction of Lean code while minimizing the trusted computing base. Each of the four operators in our probabilistic language maps directly to a small function in C++, enabling easy extraction that is sufficiently performant for practical deployment.

One advantage of working with mechanically verified implementations is that we can aggressively optimize our sampling algorithms, without worrying about introducing privacy vulnerabilities. In SampCert we prove that two Laplace sampling algorithms (originating from two different DP

developments) are both correct, allowing us to confidently switch between implementations at runtime. As we show in Section 4.2, the code we automatically extract from Lean outperforms the implementation of Canonne et al. [13] in [28].

A Mathematical Buffet. We make heavy use of Mathlib [30], Lean's comprehensive mathematics library, as many proofs require non-trivial mathematics. By tightly integrating Mathlib types into our development, we can leverage this extensive library of mathematical results to reproduce arguments from standard differential privacy proofs from the literature, instead of having to develop alternate proofs and techniques that avoid mathematical prerequisites. For example, since we are dealing with randomized algorithms we leverage results from `Mathlib.Probability` and `Mathlib.MeasureTheory`. Additionally, since we are manipulating non-trivial mathematical expressions, e.g., infinite sums, we make use of `Mathlib.NumberTheory` and `Mathlib.Topology`.

One concrete example of where this extensive library is essential was in our proof that discrete Gaussian noise establishes zCDP. In proving this on paper, Canonne et al. [13] make use of Fourier analysis and the Poisson summation formula, and we are not aware of any simple, alternate proof that avoids this technical machinery. Fortunately, Mathlib's comprehensive library already includes the relevant theorems from Fourier analysis to replicate this proof. Using Mathlib, we were able to fully formalize this zCDP result from first principles with reasonable proof effort.

Contributions. We summarize our contributions as follows:

- Mechanically verified foundations of differential privacy in Lean, utilizing Mathlib.
- Mechanically verified (discrete) Laplace and Gaussian sampling algorithms.
- A simple probability monad and novel proof techniques that streamline proof formalization.
- A Lean-code extractor used to deploy our verified algorithms at AWS.

2 PROGRAMMING WITH DIFFERENTIAL PRIVACY

In this section, we describe SampCert's generic view of differential privacy, how we can verify private mechanisms using our framework, and how we can instantiate it with different DP variants.

2.1 Abstract Differential Privacy

Differential privacy is a family of definitions that describe what it means to compute a statistic about a dataset while preserving the privacy of records in that dataset. For example, suppose that we have the genetic data from a population of consenting individuals and we want to count the number of people with some genetic mutation. Simply releasing the count is insufficient for protecting the privacy of the individuals. It has been shown time and time again that typical anonymization techniques are still susceptible to *reconstruction attacks*. For example, while birth date, gender, or postal code cannot identify an individual alone, their combinations form *quasi-identifiers* which suffice to uniquely identify 87% of the population of the Unites States [40].

Differential privacy definitions prescribe that adding or removing an individual's genetic data should not change the output statistic significantly. To achieve this, differential privacy dictates that the output statistics should be randomized. When implemented correctly, differential privacy ensures that an attacker will require access to an infeasible number of samples in order to learn any information about an individual's presence in the database.

The Mathematics of Privacy. There are several options for quantifying the privacy risk associated with releasing a randomized statistic. For our discussion, a mechanism $M : T \rightarrow PMF(Z)$ is a random function that takes an input in some domain T (typically a database), and returns a sample from a probability distribution over a countable range Z (with *probability mass function PMF(Z)*).
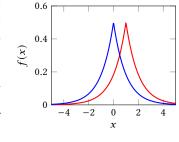
To define privacy, we fix a symmetric "adjacency" relation on T, and quantify the extent to which releasing a statistic allows an attacker to disambiguate between adjacent values of T. For example when T is the set of databases of genetic information, and databases are considered adjacent when they differ by exactly one row, differential privacy gives an upper bound on the chances that releasing a statistic leaks the genetic data of any individual—even those not included in the dataset.

In its simplest form, *pure* differential privacy says the following:

Definition 2.1 (Pure DP). A mechanism M is ϵ-DP if for all adjacent pairs $t, t' \in T$ and $S \subseteq Z$,

$$\Pr[M(t) \in S] \leq e^\epsilon \Pr[M(t') \in S] \tag{1}$$

In other words, the two probability distributions produced with t or with t' should be close enough, as specified by the *privacy parameter* $\epsilon \in \mathbb{R}$. The larger ϵ is, the weaker the privacy guarantee. This is pictorially shown in Fig. 2, with two Laplacian distributions which have different means. The closer the distributions are at each point (the smaller the ϵ) the more privacy is provided. When ϵ is small, a single random sample does not let an attacker learn much about which of the two distributions it was drawn from, preventing the attacker from learning the true value of our statistic (the mean).

Pure DP is a strong property, because an algorithm must satisfy Eq. (1) for all possible configurations S of secret information. As such, privacy researchers have developed a suite

Fig. 2. Two Laplacian distributions

of relaxations to analyze algorithms with less stringent privacy requirements. One such relaxation is *zero-concentrated* DP (zCDP) [11]:

Definition 2.2 (zCDP). A mechanism M is ρ-zCDP if for all adjacent pairs $t, t' \in T$ and $\alpha \in (1, \infty)$,

$$D_\alpha(M(t)||M(t')) \leq \alpha \rho$$

where D_α is the α-Rényi divergence between probability distributions.

In contrast to Pure DP, which gives worst-case upper bounds on the probabilities of identifying events, zero-concentrated DP gives an upper bound on the *expected privacy loss* of a mechanism. Depending on the application domain, relaxing from the *worst-case* (Pure DP) to an *average case* (zCDP) can be a realistic assumption, making zCDP a popular measure of privacy amongst researchers and in implementations of DP tools [9]. There are several other relaxations based on the expected privacy loss, such as *mean-concentrated* DP [20] and *Rényi DP* [35].

Given the wealth of privacy definitions, it is natural to try and compare their relative strengths. *Approximate* DP is an even more extreme relaxation of pure differential privacy, which allows a mechanism to violate the privacy specification with probability δ:

Definition 2.3 (Approximate DP). A mechanism M is (ϵ, δ)-DP if for all adjacent pairs $t, t' \in T$ and all subsets $S \subseteq Z$ we have

$$\Pr[M(t) \in S] \leq e^\epsilon \Pr[M(t') \in S] + \delta$$

On its own, (ϵ, δ)-DP does not provide strong privacy guarantees. A mechanism may exhibit arbitrarily insecure behavior with probability δ, such as releasing the entire secret database, and still satisfy the definition for (ϵ, δ)-DP. While the relative weakness of approximate DP makes it unsuitable for foundational proofs of privacy, it can nevertheless serve as a basis for comparing the relative strengths of privacy models. In particular, no useful definition of DP should be *weaker* than approximate DP, so all definitions of privacy should imply an approximate DP bound of some form.

```
class AbstractDP (T : Type) where
  -- A definition of Differential privacy with one monotone real-valued parameter
  -- (eg. γ-DP, γ-zCDP, etc.)
  prop : Mechanism T Z → NNReal → Prop
  prop_mono {m : Mechanism T Z} {γ₁ γ₂: NNReal} :
    γ₁ ≤ γ₂ → prop m γ₁ → prop m γ₂

  -- Privacy bound: sequential composition
  adaptive_compose_prop {U V : Type} [DiscProbSpace U] [DiscProbSpace V]
    {m₁ : Mechanism T U} {m₂ : U → Mechanism T V} {γ₁ γ₂ γ : NNReal} :
    prop m₁ γ₁ → (∀ u, prop (m₂ u) γ₂) → γ₁ + γ₂ = γ →
    prop (privComposeAdaptive m₁ m₂) γ

  -- Privacy bound: pure function postcomposition
  postprocess_prop {U : Type} [DiscProbSpace U]
    { pp : U → V } {m : Mechanism T U} {γ : NNReal} :
    prop m γ → prop (privPostProcess m pp) γ

  -- Privacy bound: constant function
  const_prop {U : Type} [DiscProbSpace U] {u : U} {γ : NNReal} :
    γ = (0 : NNReal) → prop (privConst u) γ

  -- Compatibility: Privacy parameter required to obtain (γ', δ)-approximate DP
  of_app_dp : (δ : NNReal) → (γ' : NNReal) → NNReal
  prop_app_dp [Countable Z] {m : Mechanism T Z} : ∀ (δ : NNReal) (_ : 0 < δ) (γ' : NNReal),
    (prop m (of_app_dp δ γ') → ApproximateDP m γ' δ)
```

Listing 1. Lean definition of AbstractDP. The DiscProbSpace typeclass aliases Mathlib typeclasses for a discrete probability space over a countable, inhabited type.

A Unifying Abstraction. Despite the variety in their definitions, the popular formulations of differential privacy share common characteristics pertaining to the construction and composition of private programs. Importantly, this means that the correctness of many differentially private algorithms may not depend on the precise variant of DP one may be trying to attain.

Instead of developing parallel proof frameworks to verify the different privacy bounds, in SampCert we opted for a general, abstract definition of privacy that satisfies a number of basic axioms. Using our abstract definition, we can build and reason about mechanisms that are parametric in the definition of privacy chosen. For instance, instead of separately proving privacy for a histogram mechanism for both pure and zCDP, we can construct a single proof using the abstract framework; proofs for pure and zCDP automatically follow, as they are instantiations of the abstract definition.

Listing 1 depicts the SampCert definition of *abstract differential privacy*. From a programming perspective, this typeclass outlines a domain-specific language for constructing and verifying differentially private queries over a list of natural numbers. Specifically, our abstract definition of differential privacy, which we call γ-ADP, is parameterized by a real-valued privacy parameter γ that takes on a different meaning for each instantiation (e.g., the ϵ in pure DP or the ρ in zCDP and Rényi DP). We define a set of properties that a AbstractDP instantiation must supply, and informally outline them below:

(1) **Privacy** (prop): A specification for what it means for a mechanism M to be γ-ADP, e.g., Definition 2.1 for pure DP.
(2) **Monotonicity** (prop_mono): A proof that if M is γ-ADP then it is γ'-ADP for all $\gamma' \geq \gamma$.

```
class DPNoise (dps : AbstractDP T) where
  -- A noise mechanism with sensitivity parameter (Δ) and security parameter (num/den)
  -- (eg. Discrete Laplace, Discrete Gaussian, etc.)
  noise : (query : List T → ℤ) → (Δ  : ℕ+) → (num : ℕ+) → (den : ℕ+) → Mechanism T ℤ

  -- Relationship between noise argument and privacy amount
  noise_priv : (γn : ℕ+) → (γd : ℕ+) → (priv : NNReal) → Prop
  noise_prop {q : List T → ℤ} {Δ γn γd : ℕ+} {γ : NNReal} :
    noise_priv γn γd γ →
    sensitivity q Δ →
    dps.prop (noise q Δ γn γd) γ
```

Listing 2. Lean definition of DPNoise.

(3) **Composition** (`adaptive_compose_prop`): A proof that ADP mechanisms compose additively.
(4) **Postprocessing** (`postprocess_prop`): A proof that postcomposing by functions that do not access the secret database does not degrade privacy.
(5) **Base case** (`const_prop`): A proof that constant functions are 0-ADP.
(6) **Approximate DP** (`prop_app_dp`): A proof that ADP implies approximate DP (see below).

Most of the above properties, e.g., composition and postprocessing, are standard to definitions of differential privacy. Perhaps the non-trivial one is the connection with approximate DP. Here we require that there is a function f (given by the `of_app_dp` field) with the property that, for all mechanisms M, and parameters $γ$ and $δ$, if M is $f(γ, δ)$-ADP, then M is $(γ, δ)$-DP.

Intuitively, the function f establishes a reduction between an abstract ADP and approximate DP: any $(γ, δ)$-DP privacy requirement can be satisfied by proving some ADP bound. In the pure DP case, $f(γ, δ) = γ$, as any $γ$-DP mechanism is $(γ, δ)$-DP for any $δ$. For zCDP, we use Lemma 3.5 of [12] to establish that a $ρ$-zCDP mechanism is $(ρ + \sqrt{4ρ \log(1/δ)}, δ)$-DP for any $δ$ (indeed, we were able to replicate the proof of this bound from [12] using Mathlib lemmas such as Markov's inequality and the calculus of hyperbolic trigonometry). While this property of `AbstractDP` is not used in our constructions of abstract differential privacy mechanisms, it is crucial to ensure that our instances of `AbstractDP` are consistent with respect to each other and standard models of DP.

Notes on the Lean Definitions. We walk the reader through some of the Lean definitions for clarity. Consider `prop`: it defines a proposition (`Prop`) that takes a mechanism (`Mechanism T Z`) and a real number $\mathbb{R}_{\geq 0}$ (`NNReal` in Lean), and checks if the mechanism satisfies $γ$-ADP. Now consider `prop_mono`: it defines a proposition that says if $γ_1 \leq γ_2$ and the mechanism m is $γ_1$-ADP (denoted `prop m γ_1`), then m is also $γ_2$-ADP. Consider now `adaptive_compose_prop`: it specifies that given two mechanisms that are $γ_1$-ADP and $γ_2$-ADP, composing them produces a $(γ_1 + γ_2)$-ADP (the adaptive composition function `privComposeAdaptive` is not shown).

2.2 Noise and Sensitivity

While Listing 1 describes how differentially private programs compose, it does not give us a way to obtain privacy bounds for nontrivial programs. Definitions of DP are typically presented alongside a *noise mechanism*, which prescribes a distribution and amount of noise to add to a statistic in order to obtain a given privacy bound. The amount of noise depends on both the desired privacy bound and the *sensitivity* of the statistic: statistics whose output changes greatly between adjacent databases will require more noise in order to satisfy a given privacy bound. In SampCert, we define sensitivity of a function `List T → ℤ` to be the maximum change in the value of this function obtained by adding, removing, or modifying a single entry in its input list.

```
variable {T : Type} (B : Bins T nBins) [dps : AbstractDP T] [dpn : DPNoise dps]

def privNoisedBinCount (γ₁ γ₂ : ℕ+) (b : Fin nBins) : Mechanism T ℤ :=
  (dpn.noise (exactBinCount nBins B b) 1 γ₁ (γ₂ * nBins))

def privNoisedHistogramAux (γ₁ γ₂ : ℕ+) (n : ℕ) (Hn : n < nBins) : Mechanism T (Histogram T nBins
    B) :=
  let privNoisedHistogramAux_rec :=
    match n with
    | Nat.zero => privConst (emptyHistogram nBins B)
    | Nat.succ n' => privNoisedHistogramAux γ₁ γ₂ n' (Nat.lt_of_succ_lt Hn)
  privPostProcess
    (privCompose (privNoisedBinCount nBins B γ₁ γ₂ n) privNoisedHistogramAux_rec)
    (fun z => setCount nBins B z.2 n z.1)

def privNoisedHistogram (γ₁ γ₂ : ℕ+) : Mechanism T (Histogram T nBins B) :=
  privNoisedHistogramAux nBins B γ₁ γ₂ (nBins - 1) (proof_pred_nBins_lt_nBins)
```

Listing 3. An implementation of an abstract DP histogram

Listing 2 defines a typeclass `DPNoise` that specifies the properties of an abstract noising scheme.
The typeclass is parameterized by a definition of differential privacy, and allows provers to construct
private noised statistics using the abstract noise mechanism `noise`. The `noise` function takes a rational
parameter γn/γd, a sensitivity Δ, and a Δ-sensitive query, and adds enough noise to the query to
ensure that its result is γ-ADP (`noise_prop`). We note that a noise program with arguments γn/γd
does not simply provide (γn/γd)-ADP in general (for example, the *Gaussian* noise mechanism with
arguments γn/γd satisfies $(γn/2γd)^2$-zCDP). The relationship between the desired privacy bound γ
and the function parameters γn and γd is specified in the `noise_priv` field. This setup avoids the use of
any floating point arithmetic, instead requiring that the prover either manually or programmatically
ensure that their `noise` parameters satisfy the appropriate `noise_priv` bound.[2]

2.3 Case study: Implementing and Verifying a Private Histogram

We now write and verify a simple differentially private program for computing histograms using
SampCert. In particular, we will show that the program is γ-ADP, meaning that if someone defines
a new notion of privacy as an instantiation of γ-ADP, they get a verified mechanism for free.

A *histogram* over a list is a finite vector of integers (so-called *bins*), which count how elements in
a list are assigned to each bin by some *binning function* `bin`:

```
structure Bins (T : Type) (nBins : ℕ) where
  bin : T → Fin nBins
structure Histogram (T : Type) (nBins : ℕ+) (B : Bins T nBins) where
  count : Mathlib.Vector ℤ nBins
```

Differentially private histograms are a common building block for the construction of larger
DP algorithms. For example, one can privately calculate the approximate maximum of a list by
inspecting the last inhabited bin in its histogram, an important step in calculating differentially
private means on data whose values lack tight upper bounds a priori [46]. The key idea in con-
structing a private histogram is to add enough noise to make the value in each bin (γ/nBins)-ADP;
by composition, the overall histogram will be γ-ADP.

This argument can be translated one-to-one to a verified implementation of a private histogram
in SampCert. Listing 3 depicts an abbreviated implementation of the abstract histogram function.

---

[2]Our most complicated mechanisms in SampCert required at most two proofs of `noise_priv`.

```
variable (γ₁ γ₂ : ℕ+) (γ : NNReal) (HN_bin : dpn.noise_priv γ₁ (γ₂ * nBins) (γ / nBins))

theorem exactBinCount_sensitivity (b : Fin nBins) : sensitivity (exactBinCount nBins B b) 1 := by
  rw [sensitivity]
  intros _ _ H
  cases H
  all_goals simp_all [exactBinCount, exactBinCount, List.filter_cons]
  all_goals aesop

lemma privNoisedBinCount_DP  (b : Fin nBins) :
  dps.prop (privNoisedBinCount nBins B γ₁ γ₂ b) (γ / nBins) := by
  unfold privNoisedBinCount
  apply dpn.noise_prop HN_bin
  apply exactBinCount_sensitivity
```

<div align="center">Listing 4. Proof that noised count is (γ/nBins)-DP</div>

```
lemma privNoisedHistogramAux_DP (n : ℕ) (Hn : n < nBins) :
  dps.prop (privNoisedHistogramAux nBins B γ₁ γ₂ n Hn) (n.succ * (γ / nBins)) := by
  induction n
  · unfold privNoisedHistogramAux
    simp
    apply dps.postprocess_prop
    apply dps.compose_prop (AddLeftCancelMonoid.add_zero _)
    · apply privNoisedBinCount_DP; apply HN_bin
    · apply dps.const_prop; rfl
  · rename_i _ IH
    simp [privNoisedHistogramAux]
    apply dps.postprocess_prop
    apply dps.compose_prop ?arithmetic
    · apply privNoisedBinCount_DP; apply HN_bin
    · apply IH
    case arithmetic => simp; ring_nf
```

<div align="center">Listing 5. Proving a privacy bound for privNoisedHistogramAux by induction</div>

```
lemma privNoisedHistogram_DP :
  dps.prop (privNoisedHistogram nBins B γ₁ γ₂) γ := by
  unfold privNoisedHistogram
  apply (AbstractDP_prop_ext _ ?HEq ?Hdp)
  case Hdp => apply privNoisedHistogramAux_DP; apply HN_bin
  case HEq => simp [predBins, mul_div_left_comm]
```

<div align="center">Listing 6. Top level privacy bound for privNoisedHistogram_DP</div>

Our implementation is paramaterized by a positive natural number nBins, a binning strategy B, an abstract DP system dps and abstract noising function dpn. Calculation of the overall histogram (in privNoisedHistogramAux) recursively counts and adds noise to the value for each bin using the function privNoisedBinCount. The implementation uses generic DP programs privCompose to sequence the private operations, and privPostProcess to update the resulting histogram value with the new count. This generic construction will enable a generic privacy analysis of privNoisedHistogram.

To prove that this histogram is γ-ADP, we begin by showing that calculating the noised count is (γ/nBins)-ADP. In Listing 4 we prove that the exact count in each bin has a sensitivity of 1, so that dpn.noise_prop can establish the desired DP bound.

With this result in hand, we prove a privacy bound for `privNoisedHistogramAux` by induction. The base case (an empty histogram) is handled by `dps.const_prop`. For the inductive case, the lemmas `dps.postprocess_prop` and `dps.compose_prop` break down the proof into establishing a privacy bound of (γ / `nBins`) for each bin, which we have done, and (`n` $*$ γ / `nBins`) for the recursive call, which we get from the induction hypothesis as shown in Listing 5.

Finally, in Listing 6 we establish the top-level privacy bound, which follows by simple arithmetic. This completes the privacy proof for the private histogram, giving us a result that applies to any `AbstractDP` instantiation in fewer than 50 lines of proof. In our development we use this implementation to privately compute approximate maximum and approximate mean queries, their privacy proofs are of a similar level of complexity and reuse the bound proven in Listing 6. This example demonstrates how abstract privacy reasoning is enough to verify the privacy of core DP algorithms, and that we can obtain useful privacy results without fixing a definition of DP beforehand.

2.4 Instantiating Pure DP

We now discuss how `AbstractDP` can be instantiated into standard definitions of privacy. To instantiate `AbstractDP` to pure differential privacy, we need to establish the properties outlined in Listing 1.

Defining Pure DP. We will give a flavor of some of those definitions and theorems. First, we define pure differential privacy, which is a Lean formalization of ϵ-DP from Definition 2.1:

```
def PureDP (m : Mechanism T U) (ε : ℝ) : Prop :=
  ∀ l₁ l₂ : List T, Neighbour l₁ l₂ → ∀ S : Set U,
  (Σ' x : U, if x ∈ S then m l₁ x else 0) / (Σ' x : U, if x ∈ S then m l₂ x else 0) ≤
    ENNReal.ofReal (Real.exp ε)
```

Using this definition, we can instantiate a `AbstractDP` instance for pure DP[3] that uses `PureDP` for its `prop` field. Our use of the extended nonnegative real numbers `ENNReal` means that we do not need to prove convergence of the sums in the above definition—all sums in `ENNReal` are absolutely convergent. The proofs of the `AbstractDP` properties (e.g., `adaptive_compose_prop`, `postprocess_prop`) are standard, and we refer the interested reader to our development.

Laplacian Mechanism. To provide the basic noise mechanism for pure DP, we use discrete Laplace noise, as defined below:

```
def privNoisedQueryPure (query : List T → ℤ) (Δ : ℕ+) (ε₁ ε₂ : ℕ+) (l : List T) : PMF ℤ := do
  DiscreteLaplaceGenSamplePMF (Δ * ε₂) ε₁ (query l)
```

This says: apply the Δ-sensitive query to the list `l` and add Laplacian noise with variance $\Delta\epsilon_2/\epsilon_1$. We can then show that this provides ϵ_1/ϵ_2-DP, as specified in the following theorem:

```
theorem privNoisedQueryPure_DP (query : List T → ℤ) (Δ ε₁ ε₂ : ℕ+) (bounded_sensitivity :
    sensitivity query Δ) :
  PureDP (privNoisedQueryPure query Δ ε₁ ε₂) (ε₁ / ε₂)
```

(We redact the proof as it involves close to 100 lines of code.)

Using this theorem we are able to establish the abstract properties required by `DPNoise`, and with these two instances in hand our generic proofs of privacy (e.g., Section 2.3) now establish pure differential privacy when noise is drawn using our discrete Laplace sampler.

---

[3]Note that we use a list representation for our database, and have fixed a neighboring relation `Neighbour`.

2.5 Instantiating zCDP

We now discuss how we instantiate AbstractDP to ρ-zCDP (zero-concentrated differential privacy). This notion, which has become a standard definition of differential privacy, was presented in Definition 2.2.

Recall that the definition of zCDP stipulates that $D_\alpha(M(t)||M(t')) \leq \alpha\rho$, where D_α is the α-Rényi divergence between probability distributions. In order to formalize zCDP in Lean, we also formalized the notion of Rényi divergences in Lean. Establishing the properties of ρ-zCDP is much more involved and requiring definitions of Rényi divergence, integrals, infinite sums, and Jensen's inequality. Altogether, defining an AbstractDP instance for zCDP amounts to around 3500 lines of code, however, the proofs themselves closely follow those presented by Canonne et al. [13].

Gaussian Mechanism. Analogous to pure DP, we can establish zCDP by adding noise from the discrete Gaussian distribution, as defined below:

```
def privNoisedQuery (query : List T → ℤ) (Δ : ℕ+) (ρ₁ ρ₂ : ℕ+) (l : List T) : PMF ℤ :=
    DiscreteGaussianGenPMF (Δ * ρ₂) ρ₁ (query l)
```

and establish that the result is ρ_1/ρ_2-zCDP:

```
theorem privNoisedQuery_zCDP (query : List T → ℤ) (Δ ρ₁ ρ₂ : ℕ+) (bounded_sensitivity :
    sensitivity query Δ) :
  zCDP (privNoisedQuery query Δ ρ₁ ρ₂) ((ρ₁ : NNReal) / ρ₂)
```

This theorem allows us to establish a DPNoise instance for zCDP, meaning our generic DP proofs now prove zCDP as well.

2.6 Additional Differential Privacy Results

While our AbstractDP framework is a general and powerful technique for proving DP for compositions of private programs, we remark that not every program in SampCert needs to be proven fully abstractly. In our development we establish a Pure DP bound on the *sparse vector mechanism* from Dwork and Roth [19], a program which can calculate approximate maximums using asymptotically less noise than private histograms, but to the best of our knowledge is not subject to a generic privacy analysis. Nonetheless, our development also contains a mechanization of a theorem from Bun and Steinke [11] which establishes zCDP bounds on Pure DP programs, indirectly giving us a zCDP bound on the sparse vector mechanism as well.

Notably, our AbstractDP framework does not restrict us from using programs such as the sparse vector mechanism which themselves lack a fully abstract proof. By parameterizing over a generic noised maximum function and its privacy bound, programs that use noised maximums can still be subject to the abstract privacy analysis we presented in 2.3–only proving the DP-specific privacy bounds for critical components after fixing a definition of privacy. We believe that this degree of semantic compositionality will enable SampCert to keep up with the latest developments in DP.

3 CONSTRUCTING VERIFIED SAMPLING ALGORITHMS

In this section, we discuss our implementation and proof of the *discrete* Laplace and Gaussian algorithms introduced by Canonne et al. [13], which sample from integer-valued analogues of the real-valued Laplace and Gaussian mechanisms. By using discrete mechanisms rather than rounded floating point approximations we can go beyond proofs of their privacy and provide *foundational proofs of their correctness*, describing their posterior distribution exactly.[4] We hope that this aspect of our development will be useful outside of its applications to verified differential privacy.

---

[4]To contrast, the clamping mechanism proposed by Mironov [33] privately samples from an approximation of the Laplace distribution, but to the best of our knowledge its exact posterior distribution is not known.

Verifying sampling algorithms comes with a different set of design considerations than verifying DP. In Section 2 we represented differentially private programs by a shallow embedding into Lean's `PMF` (probability mass function) type, and we saw how this allowed us to state and prove DP properties with relative ease. Unfortunately, `PMF` is not a suitable type for verifying sampling algorithms: terms in `PMF` describe only the probability mass a program takes at each point, and carry no information about how to sample from the distribution itself. Indeed, the `PMF` type lives within a `noncomputable` section in Lean; the Lean compiler does not come equipped with techniques for compiling a `PMF` into executable code.

To address this issue we introduce a new domain-specific programming language SLᴀɴɢ, which is shallowly embedded in a mass function monad SLang, and which consists solely of terms that Lean understands how to compile. Our architecture balances three needs: (1) minimizing the amount of trusted compilation code required to execute our programs, (2) allowing complete programs to behave like PMF's, and (3) supporting ergonomic reasoning principles during the intermediate stages of a correctness proof (for example, reasoning about the state of a partly unrolled loop).

3.1 A Simple, Probabilistic Language

We embed SLᴀɴɢ in the unnormalized, discrete Giry monad [23]. The general measure-theoretic version of this monad is already formalized in `Mathlib.Probability` as part of its existing probability development. This allows us to apply standard results from probability and measure theory such as Jensen's or Markov's inequality to SLᴀɴɢ programs, without redeveloping proofs for these results.

Concretely, SLang τ is the type $\tau \to \mathbb{R}_{\geq 0}^{\infty}$; it has monadic return δ. and bind $(\cdot \ggg \cdot)$ defined as

$$\delta_{v'}(v) \triangleq \begin{cases} 1 & v = v' \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

$$(p \ggg f)(v) \triangleq \sum_{t \in \tau} f(t)(v) \cdot p(t) \tag{3}$$

Formally, the series in Eq. (3) is defined as a sum inside a Mathlib topological monoid; the sum always converges absolutely since it takes values in the extended nonnegative reals. SLang terms with a proof of normalization (i.e., terms with sum 1) can be promoted into `PMF` terms, and used to instantiate our abstract DP system.

One may question as to why we express SLᴀɴɢ programs inside a mass monad with no requirements on the total mass, when we eventually require a proof that a distribution has mass 1. The principal issue arises when attempting to verify the posterior mass distribution of programs which include loops and recursion. Consider expressing an invariant on the normalized probability distribution of a program involving loops—for example, a description of the normalized mass distribution at the start of the kth iteration of the loop. Representing the precise mass function as a `PMF` would involve calculating a normalizing factor related to the conditional probability that the loop does not terminate during the first $k - 1$ iterates, a problem that involves reasoning quantitatively about how the loop body changes the mass of all possible program states changes rather than just those that are relevant for functional correctness. Indeed, such a representation requires that all loops be proven to terminate almost-surely before they are fully defined, further complicating the process.[5] We have found that such complete descriptions of loop invariants to be both uncommon in the literature and counterintuitive to derive ourselves.

---

[5]It would be particularly challenging to apply indirect methods when reasoning about the termination probability, for example by bounding the cumulative density function of the loop's posterior distribution below by a sequence that converges to 1. In our development, we used this technique in our proof of normalization for the sparse vector mechanism.

$$\text{probPure} : \tau \to \text{SLang } \tau$$
$$\text{probBind} : (\tau \to \text{SLang } \tau') \to (\text{SLang } \tau) \to \text{SLang } \tau'$$
$$\text{probUniformByte} : \text{SLang } \mathbb{N}$$
$$\text{probWhileCut} : (\tau \to \mathbb{B}) \to (\tau \to \text{SLang } \tau) \to \mathbb{N} \to \text{SLang } \tau \to \text{SLang } \tau$$
$$\text{probWhile} : (\tau \to \mathbb{B}) \to (\tau \to \text{SLang } \tau) \to \text{SLang } \tau \to \text{SLang } \tau$$

$$\text{probPure } v' \ (v) \triangleq \delta_{v'}(v)$$
$$\text{probBind } f \ p \triangleq (f \ggg p)(v)$$
$$\text{probUniformByte}(v) \triangleq \begin{cases} 2^{-8} & v < 2^8 \\ 0 & \text{otherwise} \end{cases}$$
$$\text{probWhileCut } c \ f \ n \ i \ (v) \triangleq \begin{cases} 0 & n = 0 \\ \text{probPure } i \ (v) & n > 0 \text{ and } \text{c } v = \text{false} \\ \begin{aligned} &\text{probBind} \\ &\quad (\text{probWhileCut } c \ f \ (n-1)) \\ &\quad (\text{probBind } f \ i)(v) \end{aligned} & \text{otherwise} \end{cases}$$
$$\text{probWhile } c \ f \ i \ (v) \triangleq \sup_{n \in \mathbb{N}} \ \text{probWhileCut } c \ f \ n \ i \ (v)$$

Fig. 3. SLᴀɴɢ operators and their semantics.

Fortunately, this proof burden is almost entirely avoidable when verifying programs whose PMF has a known closed form. If we are only required to prove that complete SLang programs normalize, then we are free to delay their proof of normalization until *after* we have proven their functional correctness. Ordering our proofs in this way means that the normalization proofs for our random samplers are subject to textbook analysis of well-known mathematical functions: for example, we prove that our Gaussian sampler normalizes by computing the sum of the closed form for its PMF. These arguments are straightforward to formalize in Lean, using the rich mathematical reasoning principles afforded to us by Mathlib.[6]

We can now define our programming language SLᴀɴɢ, embedded inside SLang. There are four base terms in SLᴀɴɢ (see Fig. 3) which resemble operations in an imperative probabilistic programming language. The first two constructs probPure and probBind mirror returning values and sequencing probabilistic programs, and are described using our mass function monad. The term probUniformByte is a uniform distribution over the value of a single byte, which we are able to bootstrap into uniform samples over any finite space.

The final and most complex construct probWhile is defined to be the supremum over finite truncations of a loop. In our development we prove that probWhileCut—the subdistribution obtained by truncating probWhile to a fixed number of loop iterations—has monotone increasing mass each point.[7] We call the truncation to k iterations a k *cut* of the loop. By the monotone convergence theorem, probWhile is the limit distribution of probWhileCut, as would be typical for an operational semantics of an imperative probabilistic programming language.

3.2 A Proof Technique for Loops

Enabled by our use of unnormalized mass functions, our approach to verifying probabilistic loops breaks down proofs for programs involving loops into two lemmas:

---

[6]Much like normalization, proving convergence for infinite series over arbitrary types is both difficult and unnecessary. By allowing our distributions to take values in $\mathbb{R}_{\geq 0}^{\infty}$ (a type where all series converge), we can delay the proof that they are finite everywhere past the point where we have proven functional correctness. Indeed, because our probability spaces are discrete, the fact that a PMF converges to a finite value at each point follows as a corollary of normalization.

[7]This monotonicity property would not hold for all loops if we embedded SLᴀɴɢ inside PMF.

```
def geoLoopCond (st : Bool × ℕ) : Bool := st.1

def geoLoopBody (st : Bool × ℕ) : SLang (Bool × ℕ) := do
  let x ← trial
  return (x,st.2 + 1)

def probGeometricLoop : SLang (Bool × ℕ) :=
  probWhile geoLoopCond (geoLoopBody trial) (true,0)

def probGeometric : SLang ℕ := do
  let st ← probGeometricLoop
  return st.2
```

Listing 7. An implementation of a geometric sampler in SLANG

(1) **Cut reachability**: For each possible output v, calculate the mass of v at some cut k_v.
(2) **Cut stability**: Prove that the mass at v is stable for cuts greater than k_v.

In other words, to prove that the posterior distribution of a loop has a particular closed form, we first show how each *individual point* in the sample space reaches its desired mass (reachability), and then show that the mass function is preserved for larger cuts of the loop (stability). This simplifies the limits we need to compute, and enables us to compute these limits gradually and recursively. We note that this technique does not work in the normalized setting: the normalized probability mass at an output point v may in fact *not* be stable after a fixed number of loop iterates even if it impossible for the loop to return v after that point: changing the loop's truncation alters the total probability mass, and thus the normalizing factor at each point.

3.2.1 Sampling from the Geometric Distribution. To illustrate how we reason about programs involving probWhile in SampCert we outline our verification of a sampler for the *geometric distribution*, a simple building block which will be essential in the construction of our more advanced sampling algorithms. The geometric distribution is a probability distribution on $\mathbb{N}$, parameterized by a real number $t \in [0, 1)$, and given by the PMF

$$\mathrm{Geo}_t(z) = \begin{cases} 0 & z = 0 \\ (1 - t)t^{z-1} & z > 0 \end{cases} \tag{4}$$

Informally, $\mathrm{Geo}_t(z)$ describes the probability that in a sequence of independent random events which "succeed" with probability t, the zth event is the first "failure". Listing 7 depicts a SLANG implementation of a sampler from the geometric distribution which performs this experiment. Our implementation is parameterized by a trial program from which we can draw unlimited independent and identically distributed boolean samples which are true with probability $t \in [0, 1)$. The loop state is a value of type Bool × ℕ: a tuple consisting of the result of the last trial, and the total number of trials attempted so far. The top-level program probGeometric performs the sampling experiment in a loop, and reports the total number of trials attempted once a trial fails.

We will show that the mass of probGeometric at each point $n \in \mathbb{N}$ equals the value $\mathrm{Geo}_t(n)$. To begin, a simple argument shows that probGeometricLoop never returns a state with flag true (in this case, the loop would have continued executing) so it suffices to determine the value of probGeometricLoop (false, n). Unrolling the definitions, we need to show for all $t \in [0, 1)$ and $n \in \mathbb{N}$,

$$\mathrm{Geo}_t(n) = \bigsqcup_{k \in \mathbb{N}} \text{probWhileCut geoLoopCond geoLoopBody } k \text{ (true, 0) (false, n)} \tag{5}$$

For simplicity, let us denote probWhileCut geoLoopCond geoLoopBody with the function F, whose type is $\mathbb{N} \to \mathbb{B} \times \mathbb{N} \to \mathbb{B} \times \mathbb{N} \to \mathbb{R}_{\geq 0}^{\infty}$. The value $F(k, s_i, s_f)$ is the probability mass associated to

transitioning from state s_i to state s_f using at most k (guarded) unrollings of the loop body. To prove our main result, we will use cut reachability and stability lemmas.

Cut Reachability. First, we show that *for each point n, truncating the loop to $n+1$ iterates is enough to ensure that the probability mass of sampling n is equal to* $\mathrm{Geo}_t(n)$.

We will set out to find some truncation k such that $F(k, (\mathtt{true}, 0), (\mathtt{false}, n))$ is known. The case of $n = 0$ is straightforward, $F(1, (\mathtt{true}, 0), (\mathtt{false}, 0)) = 0 = \mathrm{Geo}_t(0)$ since this program will surely increment the counter at least once. It suffices to show that for all m and k we have $F(m+2, (\mathtt{true}, k), (\mathtt{false}, m+k+1)) = \mathrm{Geo}_t(m+1)$ by specializing $m = n-1$ and $k = 0$. We can easily prove this by induction on m, thus showing that truncating the loop to $n+1$ iterates is enough to ensure that the probability mass of sampling n is equal to $\mathrm{Geo}_t(n)$.

Cut Stability. For the second part of our argument, we will show that adding extra iterates after n does not change the mass at point n. Intuitively, this is due to the fact that exactly one loop iterate can possibly terminate in state n. This can be formalized into the statement that

$$(n + 1 + k, (\mathtt{true}, 0), (\mathtt{false}, n)) = F(n + 1, (\mathtt{true}, 0), (\mathtt{false}, n))$$

and proven using a similar induction and unrolling as in the first part.

Stability + Reachability = Correctness. Put together, we have shown that for any n the sequence $\{F(i, (\mathtt{true}, 0), (\mathtt{false}, n))\}_{i \in \mathbb{N}}$ is eventually constant at value $\mathrm{Geo}_t(n)$. It is simple to prove that this sequence is monotone increasing, so the supremum of this sequence is equal to its limit, and we conclude that `probGeometric n` samples from the geometric distribution with parameter t. We formalize the argument for the normalization of Geo_t separately, and because this PMF is equal to `probGeometric` at all points, we conclude that our program is indeed normalizing.

3.2.2 Other SLang derived forms. SampCert includes verified implementations of other basic probabilistic operators which we will make use of in the following sections. For example, `probUntil` implements rejection sampling using a `probWhile` loop: a `probUntil` program repeatedly samples from a distribution until it obtains a value that satisfies a boolean predicate.

```
def probUntil (body : SLang T) (cond : T → Bool) : SLang T := do
  let v ← body
  probWhile (λ v : T => ¬ cond v) (λ _ : T => body) v
```

This allows us to bootstrap `probUniform n`, a uniform distribution over the integers $[0, n)$, by repeatedly making calls to `probUniformByte` in a `probUntil` loop. The correctness proof for `probUntil` closely mirrors the proof in Section 3.2.1, however in the *stability* part of the proof the mass has an exponential relationship to the cut value, rather than constant.

We have also formalized samplers for Bernoulli trials—in particular, a sampling algorithm `BernoulliExpNegSample` from Canonne et al. [13] for the Bernoulli trial with parameter e^{-q} where q is rational. We prove these derived forms correct using our *reachability and stability* technique, and they are an integral component for the following sampling algorithms.

3.3 The Discrete Gaussian and Laplacian

As discussed in Section 2.2, the noise distributions for Pure DP and zCDP are the *discrete Laplace* and *discrete Gaussian* distributions, respectively. In this section, we outline our implementations and proofs of those sampling algorithms. Making use of the large body of mathematical work in Mathlib, we are able to adapt mathematical arguments from the literature to prove their correctness.

```
def DiscreteLaplaceSample (p q : PNat) : SLang ℤ := do
  let r ← probUntil (DiscreteLaplaceSampleLoop p q) (λ x : Bool × Nat => ¬ (x.1 ∧ x.2 = 0))
  return if r.1 then - r.2 else r.2
```

Listing 8. An implementation the Laplace sampler using sampling loop discretelaplacesampleloop

```
def DiscreteLaplaceSampleLoop (num : PNat) (den : PNat) : SLang (Bool × Nat) := do
  let v ← probGeometric (BernoulliExpNegSample den num)
  let B ← BernoulliSample 1 2 (Nat.le.step Nat.le.refl)
  return (B, v - 1)

def DiscreteLaplaceSampleLoopIn1Aux (t : PNat) : SLang (Nat × Bool) := do
  let U ← UniformSample t
  let D ← BernoulliExpNegSample U t
  return (U,D)

def DiscreteLaplaceSampleLoopIn1 (t : PNat) : SLang Nat := do
  let r1 ← probUntil (DiscreteLaplaceSampleLoopIn1Aux t) (λ x : Nat × Bool => x.2)
  return r1.1

def DiscreteLaplaceSampleLoop' (num : PNat) (den : PNat) : SLang (Bool × Nat) := do
  let U ← DiscreteLaplaceSampleLoopIn1 num
  let v ← probGeometric (BernoulliExpNegSample 1 1)
  let V := v - 1
  let X := U + num * V
  let Y := X / den
  let B ← BernoulliSample 1 2 (Nat.le.step Nat.le.refl)
  return (B,Y)
```

Listing 9. Sampling loops for the discrete Laplace Sampler. The BernoulliExpNegSample p q function samples from a Bernoulli distribution with bias $\exp(-p/q)$, not shown here.

3.3.1 The Discrete Laplace. The Discrete Laplace distribution is a probability space over $\mathbb{Z}$, described by the PMF with a nonnegative scale parameter t.

$$\text{Lap}_t(z) = \frac{e^{1/t} - 1}{e^{1/t} + 1} \cdot e^{-|z|/t} \tag{6}$$

The discrete Laplace with *rational* scale parameter p/q can be implemented in SLang, and can serve as the noise program for a definition of pure differential privacy.

Implementing Laplace in SLang. In SampCert, we implement two different sampling algorithms from the literature, which have different performance characteristics depending on the value of the scale parameter. To simplify proofs for these samplers, we factor out parts they have in common.

Listing 8 depicts the top-level sampling procedure for the SLang discrete Laplace sampler. The program executes a *sampling loop* (DiscreteLaplaceSampleLoop) to obtain a distribution over $\mathbb{B} \times \mathbb{N}$. Samples from this space are converted into an integer, treating the boolean as the integer's sign, and resampling the value (true, 0) in order to avoid double-counting the sample at 0. If probability distribution of the sampling loop program at $(-, n)$ is $(e^{-q/p})^n \cdot (1 - e^{-q/p}) \cdot 2^{-1}$, then the distribution over $\mathbb{Z}$ after this resampling procedure will agree with $\text{Lap}_{p/q}$.

We must now implement a sampling loop procedure. Our first implementation follows the Differential Privacy Library [26], implemented in Lean by DiscreteLaplaceSampleLoop in Listing 9. This program obtains the correct sampling loop distribution using a shifted geometric sampler for

```
def DiscreteGaussianSampleLoop (num den t : PNat) (mix : ℕ) : SLang (Int × Bool) := do
  let Y ← DiscreteLaplaceSample t 1 mix
  let C ← BernoulliExpNegSample ((|Y| * t * den) - num)^2 (2 * num * t^2 * den)
  return (Y,C)

def DiscreteGaussianSample (num : PNat) (den : PNat) (mix : ℕ) : SLang ℤ := do
  let r ← probUntil (DiscreteGaussianSampleLoop num^2 den^2 (num.val / den + 1) mix) (λ x : Int ×
      Bool => x.2)
  return r.1
```

Listing 10. SampCert implementation for the discrete Gaussian sampler.

the natural number part, and an independent coin flip for the sign. By our analysis of the geometric sampler, this program both meets the correct mass distribution, and normalizes.

Our second implementation is `DiscreteLaplaceSampleLoop'` in Listing 9, based off of [13]. This implementation uses a different rejection loop in order to separate the integral and fractional parts of the geometric sampling trial, a change which substantially improves the performance of the sampler for large values of p/q. We were able to translate the proof of correctness from [13], into Lean, making use of some properties about Euclidean division from Mathlib.

Establishing Pure DP. We next prove that discrete Laplacian noise forms an instance of the `DPNoise` typeclass for pure differential privacy. In completing these proofs we use Eq. (6). Once we have this equation characterizing the PMF, our proof of DP does not need to reason explicitly about the computational parts of the algorithm such as loop invariants, and is able to apply standard arguments from the differential privacy literature directly to our Lean implementations. We conclude that the program `DiscreteLaplaceSample num den` satisfies den/num-DP.

Combining the Two Implementations of Laplace. Indeed, because the sampling loops have equal distributions, their implementation does not matter for our proofs involving `DiscreteLaplaceSample`. Figure Fig. 4 (discussed in Section 4) depicts the performance of our Laplace samplers as we increase the scale parameter. For small values of the parameter it is faster to use the first sampling loop, but for large domains it is faster to use the latter. Our top-level implementation of discrete Laplace sampling in SampCert gets the best of both worlds by dynamically switching which implementation to use at runtime. We implemented this switching far into the proof development of SampCert; the fact that all implementations of `DiscreteLaplaceSample` are *equal* meant that retrofitting our codebase with this optimization required only superficial changes to our existing privacy proofs, and significantly improved the overall performance of our samplers.

3.3.2 The Discrete Gaussian. The canonical `noise` function for zCDP is the *discrete gaussian*, a distribution over the integers with PMF $\mathcal{N}_{\mathbb{Z}}(\mu, \sigma^2)(x) = e^{-(x-\mu)^2/2\sigma^2}/N_{\mu,\sigma}$, where $N_{\mu,\sigma}$ is a normalizing constant. Our SLANG implementation (depicted in Listing 10) samples from $\mathcal{N}_{\mathbb{Z}}(0, \sigma^2)$ using a technique presented in [13], by repeatedly sampling from a Laplace distribution. Our formulation allows us to closely mirror the correctness argument from [13]. A lemma for `probUntil` ensures that the probability of sampling an integer z will be equal to the probability of `DiscreteGaussianSampleLoop` sampling $(z, true)$ divided by the normalizing constant N. Like the proof in [13], we do not require explicit reasoning about loop iterates; our lemmas for `probUntil` allow us to prove the correctness at a higher level. Adding a fixed value $\mu \in \mathbb{Z}$ to the result of this sampling procedure shifts the mean, yielding a sample from the distribution $\mathcal{N}_{\mathbb{Z}}(\mu, \sigma^2)$.

Establishing zCDP. We also prove that the discrete Gaussian function satisfies zero-concentrated differential privacy. This proof also mimics an argument presented in [13], however at a higher-level of mathematical rigor. To establish the privacy bound, we establish an upper bound on the Rényi

divergence between two samples from discrete normal distributions with different means. The key step in the argument bounds the normalizing constant for a shifted discrete Gaussian function above by the normalizing constant for the discrete Gaussian with mean zero. Proving this fact in the manner of [13] involves computing the Fourier transform of the unnormalized discrete Gaussian function, and applying the Poisson summation formula.[8] The Mathlib library contains an extensive development for verified Fourier analysis, including the Poisson summation formula, which we were able to apply directly. Altogether, the abstractions we built over SLᴀɴɢ enabled us to prove the correctness and privacy of a discrete Gaussian sampling algorithm using standard techniques from the privacy literature.

4 EXTRACTING PERFORMANT SAMPLERS FROM LEAN

Deployment of our formally verified sampling algorithms means providing an interface for projects outside of Lean to call and execute our SLᴀɴɢ samplers. Fortunately, the Lean compiler provides an extensible C++ compiler for programs written in its computable fragment. In SampCert we leverage a small trusted codebase in order to compile SLᴀɴɢ programs to C++, which can serve as an efficient intermediate language[9] suitable for deployment at scale by AWS.

4.1 Extraction and Deployment

Our simple probabilistic language, SLᴀɴɢ, has a small number of operators with direct correspondence to C++ functions. This makes extracting C++ code straightforward with a small trusted computing base (57 lines of C++). We have extracted our verified discrete sampling algorithms, which we demonstrate are more efficient than the original implementation of Canonne et al. [13] in [28]. This is partly due to the fact that we can optimize the algorithms while proving their correctness in Lean: The discrete Laplace algorithm (which is also a subroutine of Gaussian) can dynamically choose between two different versions of the algorithm, one that is more efficient for low variance and one for high variance.

Computable Functions. The Lean language makes a clear distinction between *computable* and *noncomputable* definitions. Any Lean term whose definition depends on non-computational constructs (e.g., the law of the excluded middle or classical choice) must be marked as `noncomputable`, and will be rejected by Lean's C++ compilation pipeline. As functions into the classical real numbers, the primitive four SLᴀɴɢ constructs fall into this noncomputable fragment. However, Lean provides programmers an escape hatch: In Lean, a noncomputable function can be linked against an external C++ function using an `external` annotation, allowing Lean to compile it, and thus bring it back into the computable fragment. These definitions are *external* to the logic and must be trusted. The Lean compiler can make no formal guarantees about their runtime behavior.

Therefore, it is paramount that external definitions be kept as minimal as possible. Our FFI footprint for SampCert consists of one external definition for each SLᴀɴɢ primitive, and one function to trigger top-level evaluation. In total, five functions (57 lines of C++) are enough to compile and run SLᴀɴɢ programs from Lean. Those functions, which correspond directly to SLᴀɴɢ operators, are partially shown in Listing 11. The external function for `probPure` v simply returns v. The external function for `probBind` p f obtains a value from p, and returns the value from executing f p. The external function for `probWhile` executes a C++ while loop.

Random sampling occurs in the extraction for `probUniformByte`, which reads and returns a single byte from `/dev/urandom`. Reading random bytes, rather than uniform random integers, means we do not need to include the C++ standard library in our trusted codebase and do not need to

---

[8]https://en.wikipedia.org/wiki/Poisson_summation_formula
[9]We also provide Python interface to SampCert samplers, via Python's C++ FFI. This requires an additional external function.

```
extern "C" lean_object * prob_Pure(lean_object * a, lean_object * eta) {
    lean_dec(eta);
    return a;}

extern "C" lean_object * prob_UniformByte (lean_object * eta) {
    lean_dec(eta);
    unsigned char r;
    read(urandom, &r,1);
    return lean_box((size_t) r);}

extern "C" lean_object * prob_Bind(lean_object * f, lean_object * g, lean_object * eta) {
    lean_dec(eta);
    lean_object * exf = lean_apply_1(f,lean_box(0));
    lean_object * pa = lean_apply_2(g,exf,lean_box(0));
    return pa;}
```

Listing 11. Some C++ external functions for SLᴀɴɢ operators

perform any bug-prone bit manipulation in C++. In SampCert we implement, and verify, the code which simulates uniform samples using a stream of uniformly random bytes in Lean.

4.2 Optimizations and Performance Comparison

We now demonstrate the performance of the Lean-extracted sampling algorithms compared to existing implementations. Fig. 4 shows the runtime (in ms) for different sampling algorithms as we vary the standard deviation of the Gaussian distribution. We use the discrete Gaussian (sample_dgauss) implementation supplied by Canonne et al. [13] and another implementation of the same algorithm in diffprivlib [26]. For SampCert we include three variants of the Gaussian samplers: (green) suited for higher values of the standard deviation, (yellow) suited for lower values of standard deviation, and (red) which dynamically changes its Laplace sampling algorithm based on the standard deviation, to minimize runtime. All of these are proven correct in Lean and shown in Listing 9.

As shown in Fig. 4, the extracted Lean sampler significantly outperforms the implementation of Canonne et al. [13], with more than a 2x difference in runtime. The results also show how our optimized sampler is better than either sampler individually. The sampler in diffprivlib has a runtime that is linear in the given standard deviation (blue), while our optimized sampler's runtime remains constant. The diffprivlib sampler is faster than ours for smaller values of the standard deviation. However, it uses floating point operations for calculating some of the parameters and constants used in sampling, whereas, SampCert uses rational arithmetic throughout to avoid round-off error.
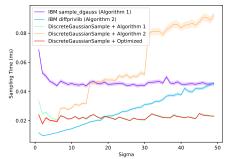

Fig. 4. Runtime of SampCert's Gaussian sampler vs other tools. The three runs of DiscreteGaussianSample are configured to use the Laplace sampling algorithm from sample_dgauss (Algorithm 1), diffprivlib (Algorithm 2), or switch between them dynamically (Optimized).

5 RELATED WORK

We now contrast our work with techniques for verifying and testing differential privacy.

Verifying DP. There is a large body of work on verifying and enforcing differential privacy. The primary differences with SampCert are the following: (1) Existing techniques are typically restricted to a specific notion of privacy, e.g., pure differential privacy, whereas SampCert aims to be generic and extensible. It is important to note however the linearly typed programming languages Duet [37] and Jazz [42], enable reasoning about a family of privacy definitions, including zero-concentrated DP. (2) Existing techniques tend to be constrained by a type system [24, 38, 45, 47], program logic [7, 8, 48], proof rules [3], or runtime system [2, 32]. These systems tend to restrict the generality of the mechanisms they can verify, in exchange for either more automation or specialized reasoning principles that simplify certain proofs. SampCert is built in Lean and admits arbitrary proofs written in Lean, extending the scope of systems that can be verified in principle, but requires manual proofs. Leveraging mathlib's extensive library helps offset this proof burden. (3) Existing techniques assume correctness of the foundations of differential privacy (e.g., composition theorems), and are taken as axioms of the type system or logic. SampCert, on the other hand, is meant as a verified foundation of DP, and therefore proves properties of DP like adaptive composition from first principles. (4) Finally, all existing techniques assume the existence of a perfect sampling algorithms. SampCert proves correctness of the sampling procedures. de Amorim et al. [14], seeking to address a similar concern over round-off errrs in trusted samplers, verify the Geometric Truncated Mechanism [6] which only requires finite precision to sample. In contrast to SampCert, they do not derive an extractable implementation based on their verification.

Automated Testing of DP. There is a body of literature focused on discovering bugs in differentially private mechanisms, instead of proofs of correctness. Various search techniques are employed, including optimization and symbolic methods. For instance, StatDP [16] uses statistical tests for counterexample generation. DP-Finder [10] transforms the search into an optimization problem using surrogate functions and numerical methods, with verification done by exact solvers like PSI [25] or sampling-based estimators. CHECKDP [44] combines verification and falsification using symbolic methods. Kolahal [39] uses the testing techniques to discover correct noise parameters for privacy mechanisms.

Reasoning about Probabilistic Loops in Shallow Embeddings. A number of prior works reason about randomized programs in theorem provers using a shallow embedding in which randomized programs are written monadically. In HOL, Hurd [27] used a state monad in which randomized programs accessed an infinite tape of uniformly sampled bits. Given such a program, a probability distribution is obtained by integrating over the space of possible tapes. Hurd [27] defines approximations of while loops that stop looping after a bounded number of iterations, analogous to our cuts, and derives a number of proof rules for deriving behaviors of unbounded loops in terms of the behavior of cuts. However, as each finite cut is still a normalized probability distribution, the stability and reachability proof technique described in Section 3.2 is not applicable. Audebaud and Paulin-Mohring [5] used a monad of *(sub)-measure transformers* in Coq. To model the semantics of randomized recursion, they axiomatize an ω-CPO structure on the interval $[0, 1]$, and derive induction principles for reasoning about fixed points. Eberl et al. [21] give a foundational definition of the Giry monad in Isabelle, which can represent general measure-theoretic programs. Other works use only a finite probability distribution monad [43]. This avoids the complexities of reasoning about countable series or measure-theoretic integration, but restricts the kinds of looping constructs that can be modeled, since general probabilistic loops can generate countable distributions.

6 CONCLUSION

We presented SampCert, a formally verified foundation for differential privacy. SampCert's key innovations include: (1) A generic DP foundation that can be instantiated for various DP definitions

(e.g., pure, concentrated, Rényi DP); (2) formally verified discrete Laplace and Gaussian sampling algorithms that avoid the pitfalls of floating-point implementations; and (3) a simple probability monad and novel proof technique that streamline the formalization. We see SampCert being used in two ways: (1) The mechanized DP foundation provides researchers with a powerful starting point for developing and proving correctness of new privacy mechanisms and definitions. (2) The mechanically verified primitives, like the random sampling algorithms, can be extracted from Lean and directly deployed to increase assurance of differentially private systems. Indeed, SampCert is used in the AWS Clean Rooms Differential Privacy service. In the future, we would like to extend SampCert to model and prove non-existence of timing side-channels. Additionally, we would like to use SampCert as to verify and extract a full-fledged DP query evaluation engine.

REFERENCES

[1] John M Abowd. 2018. The US Census Bureau adopts differential privacy. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 2867–2867.

[2] Chike Abuah, Alex Silence, David Darais, and Joseph P Near. 2021. DDUO: General-purpose dynamic analysis for differential privacy. In *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, 1–15.

[3] Aws Albarghouthi and Justin Hsu. 2018. Synthesizing coupling proofs of differential privacy. *Proceedings of the ACM on Programming Languages* 2, POPL (2018), 1–30.

[4] Apple Inc. 2017. Differential Privacy Overview. https://images.apple.com/privacy/docs/Differential_Privacy_Overview.pdf. Accessed: [2024-10-31].

[5] Philippe Audebaud and Christine Paulin-Mohring. 2006. Proofs of Randomized Algorithms in Coq. In *Mathematics of Program Construction, 8th International Conference, MPC 2006, Kuressaare, Estonia, July 3-5, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4014)*, Tarmo Uustalu (Ed.). Springer, 49–68. https://doi.org/10.1007/11783596_6

[6] Victor Balcer and Salil Vadhan. 2017. Differential privacy on finite computers. *arXiv preprint arXiv:1709.05396* (2017).

[7] Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Advanced Probabilistic Couplings for Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 55–67. https://doi.org/10.1145/2976749.2978391

[8] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Proving differential privacy via probabilistic couplings. IEEE, 1–10.

[9] Skye Berghel, Philip Bohannon, Damien Desfontaines, Charles Estes, Sam Haney, Luke Hartman, Michael Hay, Ashwin Machanavajjhala, Tom Magerlein, Gerome Miklau, Amritha Pai, William Sexton, and Ruchit Shrestha. 2022. Tumult Analytics: a robust, easy-to-use, scalable, and expressive framework for differential privacy. *arXiv preprint arXiv:2212.04133* (Dec. 2022).

[10] Benjamin Bichsel, Timon Gehr, Dana Drachsler-Cohen, Petar Tsankov, and Martin Vechev. 2018. DP-finder: Finding differential privacy violations by sampling and optimization. 508–524.

[11] Mark Bun and Thomas Steinke. 2016. Concentrated differential privacy: Simplifications, extensions, and lower bounds. In *Theory of cryptography conference*. Springer, 635–658.

[12] Mark Bun and Thomas Steinke. 2016. Concentrated Differential Privacy: Simplifications, Extensions, and Lower Bounds, Vol. 9985. 635–658. https://doi.org/10.1007/978-3-662-53641-4_24 arXiv:1605.02065 [cs.CR]

[13] Clément L Canonne, Gautam Kamath, and Thomas Steinke. 2020. The discrete gaussian for differential privacy. *Advances in Neural Information Processing Systems* 33 (2020), 15676–15688.

[14] Arther Azevedo de Amorim, Marco Gaboardi, and Vivien Rindisbacher. 2023. Verified Differential Privacy for Finite Computers.

[15] Damien Desfontaines. 2021. A list of real-world uses of differential privacy. https://desfontain.es/blog/real-world-differential-privacy.html Accessed: [2024-11-13].

[16] Zeyu Ding, Yuxin Wang, Guanhong Wang, Danfeng Zhang, and Daniel Kifer. 2018. Detecting violations of differential privacy. 475–489.

[17] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *Advances in Cryptology - EUROCRYPT 2006*, Serge Vaudenay (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 486–503.

[18] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis, Vol. 3876. 265–284. https://doi.org/10.1007/11681878_14

[19] 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. https://doi.org/10.1561/0400000042

[20] Cynthia Dwork and Guy N Rothblum. 2016. Concentrated differential privacy. *arXiv preprint arXiv:1603.01887* (2016).

[21] Manuel Eberl, Johannes Hölzl, and Tobias Nipkow. 2015. A Verified Compiler for Probability Density Functions. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science, Vol. 9032)*, Jan Vitek (Ed.). Springer, 80–104. https://doi.org/10.1007/978-3-662-46669-8_4

[22] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. 2021. Coupled relational symbolic execution for differential privacy. In *Programming Languages and Systems: 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27–April 1, 2021, Proceedings 30*. Springer International Publishing, 207–233.

[23] Roman Frič and Martin Papčo. 2010. A categorical approach to probability theory. *Studia Logica* 94 (2010), 215–230.

[24] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. 2013. Linear dependent types for differential privacy. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 357–370.

[25] Timon Gehr, Sasa Misailovic, and Martin Vechev. 2016. PSI: Exact Symbolic Inference for Probabilistic Programs. Springer International Publishing, Cham, 62–83.

[26] Naoise Holohan, Stefano Braghin, Pól Mac Aonghusa, and Killian Levacher. 2019. Diffprivlib: the IBM differential privacy library. *ArXiv e-prints* 1907.02444 [cs.CR] (July 2019).

[27] Joe Hurd. 2003. *Formal verification of probabilistic algorithms*. Technical Report UCAM-CL-TR-566. University of Cambridge, Computer Laboratory. https://doi.org/10.48456/tr-566

[28] IBM. 2020. The Discrete Gaussian for Differential Privacy. https://github.com/IBM/discrete-gaussian-differential-privacy Accessed: [2024-11-13].

[29] Jiankai Jin, Eleanor McMurtry, Benjamin IP Rubinstein, and Olga Ohrimenko. 2022. Are we there yet? timing and floating-point attacks on differential privacy systems. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 473–488.

[30] Lean. 2024. Mathlib. https://leanprover-community.github.io/mathlib-overview.html Accessed: [2024-11-13].

[31] Min Lyu, Dong Su, and Ninghui Li. 2017. Understanding the Sparse Vector Technique for Differential Privacy. 10, 6 (2017), 637–648. https://doi.org/10.14778/3055330.3055331

[32] Frank D McSherry. 2009. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. 19–30.

[33] Ilya Mironov. 2012. On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM conference on Computer and communications security*. 650–661.

[34] Ilya Mironov. 2017. Rényi differential privacy. In *2017 IEEE 30th computer security foundations symposium (CSF)*. IEEE, 263–275.

[35] Ilya Mironov. 2017. Rényi Differential Privacy. 263–275. https://doi.org/10.1109/CSF.2017.11

[36] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 theorem prover and programming language. In *Automated Deduction–CADE 28: 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings 28*. Springer, 625–635.

[37] Joseph P Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, et al. 2019. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[38] Jason Reed and Benjamin C Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. 157–168.

[39] Subhajit Roy, Justin Hsu, and Aws Albarghouthi. 2021. Learning Differentially Private Mechanisms. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 852–865. https://doi.org/10.1109/SP40001.2021.00060

[40] Latanya Sweeney. 2000. Simple Demographics Often Identify People Uniquely. (2000).

[41] Jun Tang, Aleksandra Korolova, Xiaolong Bai, Xueqiang Wang, and Xiaofeng Wang. 2017. Privacy loss in apple's implementation of differential privacy on macos 10.12. *arXiv preprint arXiv:1709.02753* (2017).

[42] Matías Toro, David Darais, Chike Abuah, Joseph P. Near, Damián Árquez, Federico Olmedo, and Éric Tanter. 2023. Contextual Linear Types for Differential Privacy. *ACM Trans. Program. Lang. Syst.* 45, 2, Article 8 (May 2023), 69 pages. https://doi.org/10.1145/3589207

[43] Eelis van der Weegen and James McKinna. 2008. A Machine-Checked Proof of the Average-Case Complexity of Quicksort in Coq. In *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5497)*, Stefano Berardi, Ferruccio Damiani, and Ugo de'Liguoro (Eds.). Springer, 256–271. https://doi.org/10.1007/978-3-642-02444-3_16

[44] Yuxin Wang, Zeyu Ding, Daniel Kifer, and Danfeng Zhang. 2020. CheckDP: An Automated and Integrated Approach for Proving Differential Privacy or Finding Precise Counterexamples.

[45] Yuxin Wang, Zeyu Ding, Guanhong Wang, Daniel Kifer, and Danfeng Zhang. 2019. Proving differential privacy with shadow execution. 655–669.

[46] Royce J Wilson, Celia Yuxin Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. 2019. Differentially private sql with bounded user contribution. *arXiv preprint arXiv:1909.01917* (2019).

[47] Danfeng Zhang and Daniel Kifer. 2017. LightDP: Towards Automating Differential Privacy Proofs. 888–901. https://doi.org/10.1145/3009837.3009884

[48] Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C. Pierce, and Aaron Roth. 2020. Testing differential privacy with dual interpreters. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 165 (Nov. 2020), 26 pages. https://doi.org/10.1145/3428233