# Machine-Checked Finite Differential Privacy in Lean

**Anonymous Authors**[1]

## Abstract

Differential privacy proofs often rely on finite-sum arguments that are routine on paper but sensitive to side conditions: events must be represented explicitly, stochastic validity is separate from the privacy inequality, and post-processing and composition rely on nonnegativity. We present a compact Lean 4 formalization of finite pure differential privacy that exposes these proof obligations theorem-by-theorem. The artifact models mechanisms as real-valued kernels, proves equivalence between pointwise and finite-event privacy, verifies post-processing and deterministic post-processing, proves two-stage adaptive and disjoint-product parallel composition theorems, and checks randomized response and a generic finite exponential mechanism. It also includes bundled stochasticity-plus-privacy preservation lemmas and a checked signed-kernel counterexample with a negative entry showing why post-processing nonnegativity cannot be dropped. The contribution is proof-engineering clarity rather than an automated theorem-proving system or a deployable privacy library. The development is intentionally limited to finite events and real-valued kernels; it does not cover approximate differential privacy, continuous distributions, verified samplers, floating-point semantics, or arbitrary interactive protocols.

## 1. Introduction

Differential privacy is a mathematical guarantee about how much the output distribution of a computation can change when the input changes by one neighboring record (Dwork & Roth, 2014). Standard textbook presentations state the guarantee over output events: for adjacent inputs $x, x'$ and any measurable event $S$, $\Pr[M(x) \in S] \leq e^{\varepsilon} \Pr[M(x') \in S]$. Many finite worked examples, however, are proved pointwise and lifted to events by summing inequalities over a finite output set. This step is elementary, but it depends on details that are often left implicit: what events are, when a table is a probability kernel, where nonnegativity is used, and what kind of composition theorem is actually being applied.

Finite-event pure DP is a useful target for formalization for three reasons. First, for stochastic finite mechanisms, event probabilities reduce to finite sums of nonnegative reals, so the algebra is visible without measure theory. Second, the side conditions matter, and they are exactly the conditions that prose proofs tend to elide; a proof assistant forces them into the open. Third, canonical finite examples, including local randomized response (Warner, 1965), use this finite-sum algebra. Getting the finite case right is therefore a self-contained step before tackling approximate DP, continuous mechanisms, or verified samplers.

This paper presents a small Lean 4 formalization of that finite-sum proof algebra, using Lean 4 (de Moura & Ullrich, 2021) and Mathlib (The mathlib Community, 2020). The contribution is not a new privacy mechanism. It is a checked account of a finite pure DP kernel calculus, designed to make routine proof obligations explicit and auditable. For AI-formath systems, the artifact can serve as a small specification target for future AI-assisted formalization studies; this paper evaluates the checked artifact itself, not an AI workflow.

The formal development contains 999 lines of Lean, 22 definitions, and 77 proved theorems. Its main components are:

- a real-valued finite-kernel model with separate nonnegativity and row-sum predicates;
- equivalence between pointwise pure DP and privacy over finite output events;
- nonnegative post-processing, deterministic post-processing, stochasticity preservation under kernel composition, and bundled stochastic-plus-private preservation lemmas;
- two-stage pair-output adaptive composition and disjoint-product parallel composition;

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

*Table 1.* Representative checked declarations.

| Declaration | Role |
|---|---|
| `pointwise_iff_event` | Pointwise/event equivalence |
| `postprocess` | Nonnegative post-processing |
| `compose_stochastic` | Stochasticity of composition |
| `adaptive_composition` | Two-stage pair-output adaptive composition |
| `exponentialMechanism_private` | Finite exponential mechanism |

- Boolean and $k$-ary randomized response; and
- a finite exponential mechanism with partition positivity, row-sum, privacy, and bundled mechanism certificates.

Table 1 previews five representative checked declarations.

The rest of the paper proceeds as follows. Section 2 fixes notation and reviews the Lean idioms used in the artifact. Section 3 presents the kernel model and privacy predicates. Section 4 states and sketches the main theorems. Section 5 records proof-engineering observations. Section 6 situates the artifact against related Lean, Isabelle, and Coq formalizations. Sections 7 and 8 state the scope of the artifact, and Appendix A describes the supplementary material.

## 2. Background and Notation

**Pure differential privacy.** Following Dwork & Roth (2014), a randomized mechanism $M : X \to \Delta(Y)$ is $\varepsilon$-differentially private with respect to an adjacency relation $\sim$ on $X$ if for every $x \sim x'$ and every measurable event $S \subseteq Y$,

$$\Pr[M(x) \in S] \leq e^{\varepsilon} \cdot \Pr[M(x') \in S].$$

The pointwise form requires the same inequality for individual outputs $y \in Y$. In a finite-event model, the two formulations coincide once events are represented as finite sets and probabilities as finite sums.

**Adjacency and neighboring databases.** The adjacency relation abstracts the usual notion of neighboring databases: two inputs are adjacent if they differ in exactly one record. The Lean development keeps the relation abstract as a function $X \to X \to \mathrm{Prop}$, so the theorems apply to any concrete neighbor relation a downstream user wants to instantiate. Symmetry is not assumed.

**Lean and Mathlib idioms.** A few idioms recur throughout the artifact and may be unfamiliar. The typeclass `[Fintype Y]` states that $Y$ has a finite enumeration and is what allows Lean to write $\sum_{y:Y}$ over the entire output type. The type `Finset Y` is the type of computable finite subsets of $Y$ and is used to model output events. Implicit binders `{x x' : X}` let Lean infer arguments from context, while strict-implicit binders, denoted `{{x x' : X}}` in Lean source, force inference to wait until a later explicit argument is supplied. The latter are used in the privacy predicates so that one can write `hM hxx' y` without naming $x$ and $x'$ explicitly.

## 3. Lean Model

The artifact models a candidate mechanism as a real-valued kernel. The base definition and the validity predicates are:[1]

```
def Kernel (X Y : Type*) := X →Y →ℝ

def Nonnegative {X Y : Type*}
    (M : Kernel X Y) : Prop :=
  ∀x y, 0 ≤M x y

def MassOne {X Y : Type*} [Fintype Y]
    (M : Kernel X Y) : Prop :=
  ∀x, ∑y, M x y = 1

def Stochastic {X Y : Type*} [Fintype Y]
    (M : Kernel X Y) : Prop :=
  Nonnegative M ∧MassOne M
```

A raw kernel is only a table of real numbers. It becomes a stochastic finite mechanism only after two separate predicates are proved: every entry is nonnegative, and each row sums to one. Keeping these predicates separate is useful because privacy inequalities, probabilistic validity, and composition theorems rely on different hypotheses. Nonnegative post-processing, for example, never uses the row-sum condition.

The development defines two privacy predicates:

```
def PureDP {X Y : Type*}
    (adjacent : X →X →Prop) (ε : ℝ)
    (M : Kernel X Y) : Prop :=
  ∀{{x x' : X }}, adjacent x x' →
    ∀y, M x y ≤Real.exp ε* M x' y

def eventMass {X Y : Type*}
    (M : Kernel X Y) (x : X)
    (s : Finset Y) : ℝ:=
  ∑y ∈s, M x y

def EventDP {X Y : Type*}
    (adjacent : X →X →Prop) (ε : ℝ)
    (M : Kernel X Y) : Prop :=
  ∀{{x x' : X }}, adjacent x x' →
    ∀s : Finset Y,
      eventMass M x s ≤
```

[1]The Lean code listings use ASCII placeholders that are rendered to math symbols by a literate map: `RR` → $\mathbb{R}$, `EPS` → $\varepsilon$, `FA` → $\forall$, `INN` → $\in$, `SUM` → $\sum$, `TIMES` → $\times$, `LBB`/`RBB` → Lean's strict-implicit binders.

```
      Real.exp ε* eventMass M x' s
```

`PureDP` is the pointwise inequality over individual outputs. `EventDP` is the corresponding inequality over finite output events, where an event is a `Finset Y`. This is a finite-event model, not a measure-theoretic model.

**Design choice.** Working with real-valued kernels rather than a more developed probability object such as a probability mass function or a measure has trade-offs. The benefit is that the algebra stays visible: theorem statements expose where finite sums and nonnegativity enter, and a reader can see that post-processing relies on nonnegativity but not on row sums. The cost is that probabilistic validity must be carried explicitly through separate predicates and preservation lemmas, rather than built in to a probability typeclass. Mathlib's `PMF` type provides probability mass functions with a baked-in validity invariant, and Mathlib's measure-theoretic probability layer provides general distributions (The mathlib Community, 2020); we use neither, because the project's purpose is to expose finite-sum side conditions, not to hide them. A larger library aimed at downstream use rather than exposition would package this structure differently.

## 4. Main Theorems

For each theorem family below, we give the Lean signature, an informal proof sketch, and a note on what side conditions Lean made explicit.

### 4.1. Pointwise/Event Equivalence

```
theorem pointwise_to_event
    {X Y : Type*} {adjacent : X →X →Prop}
    {ε : ℝ} {M : Kernel X Y}
    (hM : PureDP adjacent εM) :
    EventDP adjacent εM
```

The proof sums the pointwise inequalities $M\,x\,y \leq e^{\varepsilon} M\,x'\,y$ over the event $s$ using `Finset.sum_le_sum`, then factors the constant $e^{\varepsilon}$ out of the sum on the right with `Finset.mul_sum`. The reverse direction `event_to_pointwise` applies the event inequality to the singleton event $\{y\}$. Together these give `pointwise_iff_event`: in this finite-event model, the pointwise and event formulations are equivalent.

What Lean made explicit: the events are `Finset Y`, not arbitrary measurable sets, and `exp ε` is constant in the summation index, which is why factoring it out is legal.

### 4.2. Post-Processing

```
def compose {X Y Z : Type*} [Fintype Y]
    (M : Kernel X Y) (K : Kernel Y Z) :
    Kernel X Z :=
  fun x z => ∑y, M x y * K y z
```

```
theorem postprocess
    {X Y Z : Type*} [Fintype Y]
    {adjacent : X →X →Prop} {ε : ℝ}
    {M : Kernel X Y} {K : Kernel Y Z}
    (hM : PureDP adjacent εM)
    (hK : Nonnegative K) :
    PureDP adjacent ε(compose M K)
```

The proof multiplies each pointwise privacy inequality $M\,x\,y \leq e^{\varepsilon} M\,x'\,y$ by the nonnegative coefficient $K\,y\,z$ through `mul_le_mul_of_nonneg_right`, then sums over $y$ and pulls out $e^{\varepsilon}$. The event-level corollary `postprocess_event` is derived by routing through `pointwise_iff_event`. Deterministic post-processing is recovered by encoding a function $g : Y \to Z$ as a kernel that puts mass 1 on $g(y)$ and 0 elsewhere; the lemmas `deterministic_nonnegative`, `deterministic_mass_one`, and `deterministic_postprocess` make this a special case of the general theorem.

What Lean made explicit: the privacy half of post-processing uses only nonnegativity of $K$. Stochastic validity of $K$ (its row sums) is separately tracked by `compose_mass_one`. No theorem in this artifact treats a raw kernel as stochastic unless both predicates are supplied.

### 4.3. Composition Stochasticity

```
theorem compose_stochastic
    {X Y Z : Type*} [Fintype Y] [Fintype Z]
    {M : Kernel X Y} {K : Kernel Y Z}
    (hM : Stochastic M) (hK : Stochastic K)
        :
    Stochastic (compose M K)
```

The two halves of stochasticity are proved by separate lemmas: `compose_nonnegative` chains nonnegativity, while `compose_mass_one` swaps the order of summation, factors out $M\,x\,y$, and then uses the row-sum hypothesis on $K$ followed by the row-sum hypothesis on $M$.

What Lean made explicit: this theorem requires `[Fintype Y]` and `[Fintype Z]`, because the proof iterates the swap-and-factor identity twice. A measure-theoretic version would need a different infrastructure entirely.

### 4.4. Two-Stage Pair-Output Adaptive Composition

The adaptive construction first samples $y$ from $M$, then chooses a second kernel $N\,y$ as a function of $y$, and outputs the pair $(y, z)$. Concretely:

```
def adaptivePair {X Y Z : Type*}
    (M : Kernel X Y) (N : Y →Kernel X Z) :
    Kernel X (Y ×Z) :=
  fun x yz => M x yz.1 * N yz.1 x yz.2

theorem adaptive_composition
```

```
    {X Y Z : Type*}
    {adjacent : X →X →Prop}
    {ε1 ε2 : ℝ}
    {M : Kernel X Y} {N : Y →Kernel X Z}
    (hM : PureDP adjacent ε1 M)
    (hN : ∀y, PureDP adjacent ε2 (N y))
    (hMnonneg : Nonnegative M)
    (hNnonneg : ∀y, Nonnegative (N y)) :
    PureDP adjacent (ε1 + ε2)
      (adaptivePair M N) := by
  intro x x′ hxx′ yz
  unfold adaptivePair
  have hMstep :
     M x yz.1 ≤Real.exp ε1 * M x′ yz.1 :=
    hM hxx′ yz.1
  have hNstep :
     N yz.1 x yz.2
       ≤Real.exp ε2 * N yz.1 x′ yz.2 :=
    hN yz.1 hxx′ yz.2
  calc
    M x yz.1 * N yz.1 x yz.2
       ≤(Real.exp ε1 * M x′ yz.1)
          * (Real.exp ε2 * N yz.1 x′ yz.2)
             := by
        exact mul_le_mul hMstep hNstep
          (hNnonneg yz.1 x yz.2)
          (mul_nonneg
            (le_of_lt (Real.exp_pos ε1))
            (hMnonneg x′ yz.1))
    _ = Real.exp (ε1 + ε2)
        * (M x′ yz.1 * N yz.1 x′ yz.2) :=
            by
        rw [Real.exp_add]; ring
```

The proof is the standard product-of-inequalities argument: multiply the first-stage and second-stage privacy bounds, then collapse $e^{\varepsilon_1}e^{\varepsilon_2}$ to $e^{\varepsilon_1+\varepsilon_2}$ using `Real.exp_add`. The event-level corollary `adaptive_composition_event` is again derived through the equivalence theorem. Companion lemmas `adaptivePair_nonnegative`, `adaptivePair_mass_one`, and `adaptivePair_stochastic` carry validity through the construction independently of privacy.

What Lean made explicit: `mul_le_mul` requires nonnegativity of both right-hand factors, which is why both `hMnonneg` and `hNnonneg` appear in the statement. They are separate hypotheses from privacy, and the theorem cannot be applied without them. This theorem is a finite two-stage construction; it is not a theorem about arbitrary interactive protocols, and we are careful not to call it that.

### 4.5. Boolean Randomized Response

The Boolean mechanism is parameterized by truthful and flipped probabilities and specialized at the end to $(3/4, 1/4)$:

```
def binaryResponse (truth flip : ℝ) :
   Kernel Bool Bool :=
  fun x y => if x = y then truth else flip

theorem binaryResponse_private
   {truth flip ε: ℝ}
   (heps : 0 ≤ε)
   (htruth : 0 ≤truth) (hflip : 0 ≤flip)
   (htruth_flip : truth ≤Real.exp ε* flip)
   (hflip_truth : flip ≤Real.exp ε* truth)
       :
   PureDP (fun _ _ : Bool => True) ε
     (binaryResponse truth flip)

def randomizedResponse34 : Kernel Bool
    Bool :=
  binaryResponse ((3 : ℝ) / 4) (1 / 4)

theorem randomizedResponse34_private :
   PureDP (fun _ _ : Bool => True) (Real.
       log 3)
     randomizedResponse34
```

The general theorem is proved by case-splitting on the four input/output Boolean pairs with `fin_cases` and resolving each by the relevant cross-inequality. The specialization at $(3/4, 1/4)$ uses $e^{\log 3} = 3$ to discharge the cross-inequalities through `norm_num`. The artifact verifies privacy at $\log 3$; it does not separately prove a lower-bound theorem, although the displayed table has same-bit/flip ratio 3.

What Lean made explicit: the four side conditions `htruth_flip`, `hflip_truth`, `0 ≤ truth`, and `0 ≤ flip` are all needed in the general statement. The $\varepsilon = \log 3$ specialization follows from $e^{\log 3} = 3$, an identity that requires its own positivity side condition.

### 4.6. $k$-ary Generalization

The Boolean construction is the simplest instance of an "all-equal-flip" $k$-ary family. The artifact also formalizes `kResponse k truth flip` for finite output type `Fin k`, with truthful answer probability `truth` and a uniform flip probability for each of the $k-1$ incorrect outputs:[2]

```
def kResponse (k : Nat)
   (truth flip : ℝ) :
   Kernel (Fin k) (Fin k) :=
  fun x y => if x = y then truth else flip

theorem kResponse_private
   {k : Nat} {truth flip ε: ℝ}
   (heps : 0 ≤ε)
   (htruth : 0 ≤truth) (hflip : 0 ≤flip)
   (htruth_flip : truth ≤Real.exp ε* flip)
```

[2]The privacy lemma stated below needs no additional typeclass on $k$. The companion stochasticity lemma `kResponse_stochastic` additionally requires `[NeZero k]` so that the row-sum identity $\texttt{truth} + (k-1) \cdot \texttt{flip} = 1$ is well-defined. The cases $k = 0$ and $k = 1$ are vacuous or degenerate; the randomized-response interpretation is the nontrivial $k \geq 2$ case. See `Extensions.lean` for the full statements.

```
    (hflip_truth : flip ≤Real.exp ε* truth)
        :
  PureDP (fun _ _ : Fin k => True) ε
    (kResponse k truth flip)
```

The proof reuses the Boolean shape: case-split on whether the input matches the output, and discharge each of the four resulting cases from the four hypotheses. The structural change is that `fin_cases` enumeration over `Bool` is replaced by two nested `by_cases` on equality, since `Fin k` no longer has a fixed two-element enumeration. Stochasticity follows from a row-sum identity $\texttt{truth} + (k-1) \cdot \texttt{flip} = 1$, and the event-level corollary `kResponse_event_private` is derived through `pointwise_to_event` as before. As a worked specialization, `k3Response` verifies the $k = 3$ case at level $\log 2$.

The extension file also formalizes a finite exponential mechanism. Given a finite output type, score function $q : X \to Y \to \mathbb{R}$, and scale $\eta$, `exponentialMechanism` normalizes weights $\exp(\eta q(x, y))$ by a finite partition function. Lean proves the partition function is positive for nonempty finite output types, the normalized kernel has row sum one, and a symmetric score-sensitivity condition implies `PureDP` at budget $2\eta\Delta$. The bundled `exponentialMechanism_privateMechanism` theorem additionally records the nonnegative-budget and stochasticity side conditions.

What Lean made explicit: the side-condition tracking generalizes verbatim from $k = 2$. The same four hypotheses appear in the same roles, and the $\log 2$ specialization reduces to the same `norm_num` discharge once the cross-inequalities are made concrete. For the privacy theorem itself, generalizing the output type does not introduce a new inequality hypothesis; it only replaces `fin_cases` over `Bool` with `by_cases` on equality.

## 5. Proof-Engineering Observations

The artifact is intentionally small. Its value is that it separates proof obligations that prose proofs frequently conflate:

- *Raw kernels versus stochastic mechanisms.* Reusing a kernel as a probability distribution requires nonnegativity *and* row-sum, and proofs of one half do not give the other. No theorem in the artifact treats a raw `Kernel X Y` as a probability mechanism without both.
- *Pointwise inequalities versus event inequalities.* These are propositionally equivalent in the finite setting, but they require different proof shapes (sum-of-inequalities versus singleton specialization). A formal statement has to commit to one of them in each lemma.
- *Finite events versus finite output spaces.* The fact that an event is a `Finset Y` is independent of `[Fintype Y]`: post-processing requires the latter to define `compose`, while event privacy requires the former to state the inequality. Conflating them collapses two distinct hypotheses.
- *Privacy preservation versus probabilistic validity.* The post-processing privacy theorem needs only nonnegativity of the inner kernel; preserving stochasticity of the composite requires the row-sum condition as a second, separate lemma. Carrying these together creates spurious dependencies in downstream proofs.
- *Two-stage pair-output composition versus full interactive protocols.* The Lean theorem is exactly what it says: a two-stage construction in which the second kernel may depend on the first output. Calling it "adaptive composition" without that qualifier is a category error and would mislead a reviewer about scope.

These distinctions matter because privacy arguments are frequently reused compositionally. Lean forces every hypothesis needed by those reuse steps to appear in the theorem statement, which prevents applying these Lean lemmas without the required hypotheses.

**A concrete side-condition example.** The textbook proof of post-processing closure in Dwork & Roth (2014), Section 2.3, expands $\Pr[f(M(x)) \in S] = \sum_r \Pr[f(r) \in S] \cdot \Pr[M(x) = r]$ and applies the privacy inequality termwise to obtain $\sum_r \Pr[f(r) \in S] \cdot e^{\varepsilon} \Pr[M(x') = r]$. The inequality step uses $\Pr[f(r) \in S] \geq 0$ and $\Pr[M(x') = r] \geq 0$ implicitly, since flipping signs on either factor would invert the inequality at that index. The Lean artifact replaces $\Pr$ with a real-valued kernel, so the implicit nonnegativity becomes an explicit hypothesis: `postprocess` requires `Nonnegative K`, and the theorem cannot be applied to a kernel until that predicate is proved. The artifact also proves `negativePostprocess_mass_one` and `negative_postprocess_can_break_privacy`: the signed post-processing kernel has each row summing to 1, but because one entry is $-1$, composing it with a valid $\log 2$-private Boolean kernel violates the target privacy inequality. Thus row-sum validity alone is not enough. The same pattern recurs in `adaptive_composition`, which forces nonnegativity on both $M$ and each $N\,y$ before `mul_le_mul` closes. This is the kind of side condition that prose obscures by leaving probabilities implicit.

Table 2 maps each obligation a textbook proof routinely asserts to the corresponding Lean hypothesis and the theorems where the hypothesis is consumed. The mapping is direct: each row records a hypothesis or typeclass assumption consumed by at least one checked theorem.

*Table 2.* Paper proof obligations and their Lean hypotheses (theorems consuming each hypothesis shown in italics).

| Prose obligation | Lean hypothesis (theorems) |
|---|---|
| Output sums equal to one | `MassOne M`, `MassOne K` *(compose_mass_one, compose_stochastic)* |
| Nonnegative weights | `Nonnegative K`; `Nonnegative M`; $\forall y$, `Nonnegative (N y)` *(postprocess, adaptive_composition)* |
| Adjacency relation | `adjacent x x'` *(PureDP, EventDP)* |
| Privacy budget nonnegative | $0 \leq \varepsilon$ *(binaryResponse_private)* |
| Privacy ratio cross-inequalities | `truth <= exp eps *` `flip` and converse *(binaryResponse_private, kResponse_private)* |
| Finite output enumeration | `[Fintype Y]`, `[Fintype Z]` *(compose_stochastic)* |

## 6. Related Work

The closest Lean comparison is SampCert, a substantially broader Lean 4 foundation for differential privacy that includes verified discrete and continuous samplers, $(\varepsilon, \delta)$-DP, and a richer probabilistic infrastructure (de Medeiros et al., 2025). The present work is a deliberately minimal algebraic core for the finite pure-DP fragment: it does not aim to subsume SampCert and would not be useful as a sampler library. SampCert builds broad DP foundations; this artifact isolates the finite-sum kernel calculus and exposes every side condition as a separate theorem hypothesis. There are no hidden probability abstractions; the algebra is visible by design.

Isabelle/HOL work by Sato & Minamide (2025) formalizes pure and approximate DP for discrete and continuous distributions, including Laplace noise and report-noisy-max-style mechanisms. CertiPriv, by Barthe et al. (2013), develops a Coq program logic for relational reasoning about privacy and applies it to a range of mechanisms. Both projects target a more general setting than the finite-event algebra considered here. Beyond proof assistants, program-logic and type-system frameworks such as apRHL (Barthe et al., 2016) and Fuzz (Reed & Pierce, 2010) verify DP for executable programs through approximate relational logics and sensitivity-typed languages; these target executable mechanisms rather than the kernel-checked statements presented here.

In a different direction, Balcer & Vadhan (2019) argue that finite-computer constraints (precision, bounded representation) introduce real privacy issues that idealized analyses can miss, and Rindisbacher et al. (2023) formalize a Coq account of fixed-precision DP. That line of work targets implementation-level concerns and is complementary to ours: our finite model is about finite event spaces and finite-sum proof algebra, not about machine representation or executable sampling.

Among Mathlib components, the probability mass function library `Mathlib.Probability.ProbabilityMassFunction` (PMF) and the measure-theoretic probability files (The mathlib Community, 2020) provide more developed probability objects but are not used here, because abstracting them away would defeat the side-condition-exposure goal.

Where this artifact sits: it has the smallest scope of the projects in Table 3, and that is the methodological choice. The finite real-valued kernel core is the deliberately minimal algebraic core; every nonnegativity, row-sum, and event hypothesis appears in a theorem statement rather than being absorbed into a richer probability typeclass. The artifact's mechanism coverage extends beyond Boolean to a $k$-ary family (Section 4.6), but it remains a proof-engineering case study, not a competing infrastructure.

## 7. Limitations

The artifact does not cover approximate DP, continuous distributions, measurable events, verified samplers, random-number generation, floating-point arithmetic, implementation-level guarantees, or arbitrary interactive protocols. The adaptive composition theorem is a finite two-stage pair-output result; it should not be cited as a general adaptive composition theorem. The artifact is also not an evaluation of automated theorem proving: it does not report prompt logs, model success rates, compile traces, or workflow studies, and its contribution is the finished formalization rather than the process by which it was produced.

## 8. Conclusion

The artifact is a Lean 4 formalization of the finite-sum algebra behind pure differential privacy, comprising 22 definitions, 77 theorems, and 999 lines across the two final-project proof files. It checks the equivalence of pointwise and finite-event privacy, nonnegative and deterministic post-processing, stochasticity of kernel composition, bundled stochasticity-plus-privacy preservation, two-stage pair-output adaptive composition with additive budget, disjoint-product parallel composition, Boolean randomized response at $\log 3$, a $k = 3$ instance of $k$-ary randomized response at $\log 2$, a finite exponential mechanism, and a row-sum-one signed-kernel counterexample showing why post-processing nonnegativity matters. The artifact does not replace SampCert or report an automated theorem-proving experiment. Its value is that it makes the side conditions in textbook DP proofs auditable: finite events, nonnegativity, row sums,

*Table 3.* Related machine-checked DP formalizations: this work has the smallest scope by design and emphasizes explicit side-condition tracking. "Samplers" refers to formal correctness of executable randomness; "Approx. DP" refers to $(\varepsilon, \delta)$-DP or stronger relaxations.

| Project | Assistant | Distributions | Samplers | Approx. DP |
|---|---|---|---|---|
| SampCert (de Medeiros et al., 2025) | Lean 4 | Discrete + continuous | Yes | Yes |
| Sato/Minamide (Sato & Minamide, 2025) | Isabelle/HOL | Discrete + continuous | Partial | Yes |
| CertiPriv (Barthe et al., 2013) | Coq (relational) | Discrete + continuous | No | Yes |
| Rindisbacher et al. (Rindisbacher et al., 2023) | Coq | Finite computers | Fixed-precision | No |
| **This work** | Lean 4 | Finite kernels | No | No |

stochastic validity, and the gap between a privacy inequality and a valid probability kernel each appear as their own hypothesis or their own lemma.

## References


Balcer, V. and Vadhan, S. Differential privacy on finite computers. *Journal of Privacy and Confidentiality*, 9(2), 2019. doi: 10.29012/jpc.679.

Barthe, G., Köpf, B., Olmedo, F., and Zanella-Béguelin, S. Probabilistic relational reasoning for differential privacy. *ACM Transactions on Programming Languages and Systems*, 35(3):9:1–9:49, 2013. doi: 10.1145/2492061.

Barthe, G., Gaboardi, M., Grégoire, B., Hsu, J., and Strub, P.-Y. Proving differential privacy via probabilistic couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS 2016)*, pp. 749–758, 2016. doi: 10.1145/2933575.2934554.

de Medeiros, M., Naveed, M., Lepoint, T., Kahsai, T., Ravitch, T., Zetzsche, S., Joshi, A., Tassarotti, J., Albarghouthi, A., and Tristan, J.-B. Verified foundations for differential privacy. *Proceedings of the ACM on Programming Languages*, 9(PLDI):1094–1118, 2025. doi: 10.1145/3729294.

de Moura, L. and Ullrich, S. The Lean 4 theorem prover and programming language. In Platzer, A. and Sutcliffe, G. (eds.), *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pp. 625–635. Springer, 2021. doi: 10.1007/978-3-030-79876-5_37.

Dwork, C. and Roth, A. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3–4):211–487, 2014. doi: 10.1561/0400000042.

Reed, J. and Pierce, B. C. Distance makes the types grow stronger: A calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*, pp. 157–168, 2010. doi: 10.1145/1863543.1863568.

Rindisbacher, V., de Amorim, A. A., and Gaboardi, M. Verified differential privacy for finite computers. In *The 9th International Workshop on Coq for Programming Languages (CoqPL 2023)*, 2023. Extended abstract.

Sato, T. and Minamide, Y. Formalization of differential privacy in Isabelle/HOL. In *Proceedings of the 14th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2025)*, pp. 67–82, 2025. doi: 10.1145/3703595.3705875.

The mathlib Community. The Lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*, pp. 367–381, 2020. doi: 10.1145/3372885.3373824.

Warner, S. L. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965. doi: 10.1080/01621459.1965.10480775.


## A. Artifact Appendix

**Toolchain.** The artifact is developed against Lean `leanprover/lean4:v4.27.0` with Mathlib pinned through `lake-manifest.json`. The toolchain is fixed in the file `lean-toolchain` at the project root.

**Build instructions.** From the project root:

```
lake exe cache get
lake build
```

The first command pulls the Mathlib build cache; the second compiles the project. A successful build with no kernel-checked errors is the verification target. The supplement also includes a Dockerfile for reviewers who prefer a containerized build.

**Theorem inventory.** Table 4 lists ten representative theorems and their roles. The full list of 77 theorems is in the source and generated supplement inventory.

*Table 4.* Representative theorem inventory.

| Theorem | Role |
|---|---|
| `pointwise_to_event` | Pointwise to event |
| `event_to_pointwise` | Event to pointwise |
| `pointwise_iff_event` | Equivalence |
| `postprocess` | Pointwise post-processing |
| `postprocess_stochastic_private` | Bundled validity/privacy |
| `deterministic_postprocess` | Deterministic special case |
| `compose_stochastic` | Composition stochasticity |
| `parallel_composition` | Product parallel composition |
| `negative_postprocess_can_break_privacy` | Nonnegativity counterexample |
| `exponentialMechanism_private` | Finite exponential mechanism |

**File structure.** The supplementary material contains the Lean source (under an anonymized package directory), `lean-toolchain`, `lakefile.toml`, `lake-manifest.json`, a `README`, and the theorem-inventory table. There are no scripts, datasets, or model logs to include, because the contribution is a finished formalization.

**Audit command.** The supplement includes a no-placeholder audit:

```
rg -n "sorry|admit|axiom|unsafe" \
   Artifact.lean Artifact/
```

This command searches the shipped Lean artifact for the four kinds of escape hatch a Lean development can use to bypass kernel checking: `sorry` (admit a goal), `admit` (the same), `axiom` (introduce an unproved fact), and `unsafe` (skip safety checks). A clean run reports zero hits in the submitted artifact. Mathlib and non-submission course or experiment files are excluded from this audit.

**Anonymized supplement.** For double-blind review, the supplement uses neutral names and contains no public repository links, author identifiers, institution names, or course identifiers. Local file paths in the supplement are relative to the project root.