Formal Verification of OpenDP Transformations

Parker J. Rule

May 3, 2022

Abstract

We explore refinements of the proof method used for the verification of transformations (deterministic functions on datasets) in the OpenDP differential privacy library. In particular, we translate existing OpenDP Rust code to a format suitable for automated reasoning using the Lean theorem prover and demonstrate that key correctness properties—including domain-metric compatibility, output domain correctness, and stability relation correctness—can be proven rigorously using Lean. We argue that this method, though still reliant on a manual translation step, is more precise (and often easier) than the pseudocode-based method used in the OpenDP whitepapers project.

Contents

1 Introduction

1.1 The need for verified differential privacy frameworks

Real-world implementations of differentially private systems are vulnerable to a variety of attacks that are not theoretically obvious. Most presentations of classical DP mechanisms (for instance, [DR13]) assume a computation model with infinite numerical precision and unbounded computation time; these assumptions are at odds with the finitudes of hardware. This friction is the source of a number of creative and devastating ways to break DP guarantees. These vulnerabilities include *timing attacks*, where irregularities in how long supposedly differentially private queries take to complete may inadvertently reveal information about the private database [HPN11; Mir12; BV19]; and *floating-point irregularity attacks*, where implementations of DP mechanisms that depend on samples from continuous distributions are susceptible to subtleties in finite-precision floating-point arithmetic [Mir12].

Though floating-point arithmetic is a source of subtle bugs, DP implementers must also be careful to avoid issues that may occur in finite field arithmetic. Consider the example of a DP sum mechanism over k-bit integers using an accumulator with $k' \geq k$ bits. If not implemented carefully, it is possible for the private sum to overflow from the maximum representable k'-bit integer to the minimum representable k'-bit integer, even when the constituent integers in the sum are bounded within a relatively small range. The sensitivity of this implemented mechanism is much larger than we might theoretically expect, so noise addition based on the theoretical sensitivity may be insufficient to guarantee that the implementation is DP. Within numerical code, there is potential for "off by a sign"–type errors and other basic programming mistakes that can only be detected with statistical benchmarks.

One promising way to address these issues is the development of trusted core libraries for differential privacy. Analysts who wish to craft domain-specific differentially private analyses can compose deterministic *transformations* (such as the sum) to aggregate or otherwise postprocess their private datasets with randomized *measurements* (such as the Laplace mechanism) that ensure the released analyses satisfy the desired privacy guarantees.[1] The powerful composition theorems in the DP literature can be used to show that such a pipeline is differentially private under the assumption that its components are properly implemented. These individual components can then be vetted by a centralized organization, possibly with the aid of formal methods such as model checking and automated deduction.

1.2 Verification techniques

Within the programming languages community, there has been a significant effort to design powerful type systems that yield formal differential privacy

[1] We use the transformation/measurement nomenclature of the OpenDP framework throughout this paper; measurements are often referred to in the literature as *noise mechanisms*.

guarantees—that is, programs written in languages that use these type systems are differentially private *by construction* [RP10; ZK17; Nea+19; ADN21]. While the guarantees achieved are impressive—for instance, [AH18] describes an automated proof of the sparse vector mechanism, which is notably finicky to verify by hand [LSL16]—these programming languages are quite specialized and have not seen broad industry use. It has been noted in [ADN21] that many DP languages rely on type systems that do not exist in mainstream languages like Rust—or even popular research languages like Haskell. As a corollary, most programmers are unfamiliar with the concepts used in DP languages (though work on proving differential privacy in Hoare logic [Bar+14] suggests that integrating proofs of DP into general-purpose imperative languages may be feasible). Furthermore, most DP languages are tightly coupled to specific notions of differential privacy—a type system that yields ϵ–DP guarantees cannot necessarily be generalized to incorporate zCDP or (ϵ, δ)–DP.

Popular differential privacy frameworks [JNS18; Joh+20; Pro22a; Pro22b; Goo22] typically use a general-purpose language such as C++, Java, or Rust. There has been relatively little work on the formal verification of specific DP implementations that use non-idealized computer arithmetic and do not rely on specialized programming languages, though the pathological effects of floating-point arithmetic on general DP implementations are well-studied [Mir12; BV19; Wag22]. However, there is an extensive body of work on the formal verification of other kinds of programs. For instance, the seL4 microkernel is guaranteed to never crash or perform an unsafe operation [Kle+09]. Proving the correctness of large programs is an infamously daunting task, but it is feasible with modern tooling such as Lean [Mou+15] and Isabelle/HOL [NWP02] to verify the correctness of isolated functions of the size common within DP libraries.

We note that the task of "verifying correctness" is ambiguous. DP-oriented type systems can be used to track compositionality guarantees, but they do not necessarily solve the timing and precision issues described in section 1.1. Here, we focus our analysis on functions with finite integer input domains and output domains and ignore timing issues.

1.3 OpenDP

The *OpenDP* framework is a collection of differential privacy transformations and measurements (coupled with statistical samplers) that is designed for "black-box" use by analysts. Recent work by members of OpenDP (see the whitepapers project) has focused on verifying the correctness of OpenDP's transformations, at least for integer data types. Here, we paraphrase a few key definitions used in these initial proofs [Tea20; CTW21a; CTW21b].

Definition 1.1 (Transformation)**.** A *transformation* is a deterministic mapping between datasets. Transformations are characterized by an input domain, input metric, output domain, output metric, and stability relation.

The *stability* of a transformation is determined by how much its output changes when its input is perturbed.

Definition 1.2 (Stability relation)**.** Given some distance d_{in} between two input datasets and a candidate distance d_{out} between a transformation's outputs on these two inputs, a stability relation $\mathsf{rel}(d_{\text{in}}, d_{\text{out}})$ is a Boolean function that indicates whether d_{out} is at least the worst (largest) possible output distance.

This definition does not precisely specify how distances between inputs and outputs should be measured. The choice of input metric must type-compatible with its input domain. For instance, if a transformation takes vectors of input, a distance metric suitable for measuring the distance between pairs of vectors must be used.

Definition 1.3 (Symmetric distance between vectors)**.** Let $v_1, v_2 \in \mathcal{V}$, where $\mathcal{V}$ is a finite-dimensional vector space. Let m_1, m_2 be multisets containing the entries of v_1 and v_2 (so if v_1 and v_2 are equivalent up to element order, then m_1 and m_2 are equivalent). Then the *symmetric distance* between v_1 and v_2 is the cardinality of the symmetric difference between m_1 and m_2, commonly written as $|m_1 \Delta m_2|$.

Claim 1.1. Symmetric distance is a metric. (See [CTW21a] for details.)

Definition 1.4 (see [Ope])**.** A *vetted transformation* in OpenDP must satisfy the following properties:

(a) The transformation's metrics are compatible with its domains.

(b) The transformation emits a value in its specified output domain when given a value in its input domain.

(c) For a given $(d_{\text{in}}, d_{\text{out}})$ pair, the transformation's stability relation returns `false` if d_{out} is not an upper bound for the transformation's output distance given two datasets with distance d_{in}.

Draft proofs in the OpenDP `whitepapers` project [21] show that Definition 1.4 holds for a transformation by manually analyzing the transformation's Rust code, usually with a pseudocode translation step. We will study how to achieve similar guarantees with the aid of interactive theorem proving.

1.4 The Lean theorem prover

Interactive theorem proving systems such as Coq and Lean allow for the precise encoding of definitions, theorems, and proofs in a machine-readable representation. These systems, which are typically bootstrapped from a small number of type-theoretic axioms, are designed to interactively guide mathematicians through the construction of a proof with the aid of *proof tactics* (metaprograms that attempt to carry out a series of otherwise tedious proof steps). Due to the level of logical precision they require, interactive theorem proving systems

are often difficult to use for beginners—proof gaps that most mathematicians would be content to elide in a pen-and-paper proof must be filled in. With these difficulties, however, come formal guarantees of correctness, and systems like Coq have been used to produce machine-checked proofs of famously challenging results like the four-color theorem [Gon08] and the odd order theorem [Gon+13].[2]

These systems can also be used to verify properties of programs. Coq and Lean are both theorem provers and full-fledged functional programming languages; they leverage the *Curry–Howard correspondence* between proofs and programs to allow reasoning about properties of programs. We will show that Lean translations of OpenDP transformations satisfy the properties in Definition 1.4 using these techniques. We chose Lean (specifically, version 3) due to its rich support for *dependent types*, which are useful for encoding OpenDP input and output domains. Lean 4, which is currently under development, will support foreign function interface (FFI) calls, allowing for calling formally verified Lean code from a general-purpose programming language.

2 Approach

2.1 Proof by transpilation

The whitepapers project uses an informal translation technique, which we will call "proof by transpilation", to reason about OpenDP: Rust code is manually mapped to Python-like pseudocode. The full semantics of this pseudolanguage are not formally specified (though the behavior of many individual functions has been specified in English), so instead the readers of these proofs are left to informally interpret the code's behavior. In most cases, this imprecision is not a serious issue, as the expressions in the pseudocode are fairly straightforward. However, the cumulative divergences between code and pseudocode may affect the correctness of the proof.

As an example, pseudocode used in the proof for the `clamp` transformation is reproduced in Figure 2 next to the original Rust code in Figure 1. Observe that there are several semantic differences between the two versions of the function. For instance, the pseudocode skips an initial check of the relationship between the `lower` and `upper` parameters, and therefore the return types of the two versions differ. Some details of Rust type coercion and memory management (note the use of the `move` keyword and the `.collect()` and `.clone()` methods) are elided in the pseudocode. The implementation of element-level clamping is specified in the pseudocode but deferred to a separate `clamp` function in the Rust code. These are all reasonable simplifications—the core meaning of the original Rust is certainly preserved—but some information is lost. This is compounded by the lack of a machine-readable specification of the pseudocode's behavior.

[2] For an informal introduction to the merits of interactive theorem proving in general mathematics, see [Mic19].

```rust
pub fn make_clamp_vec<M, T>(lower: T, upper: T)
-> Fallible<Transformation<VectorDomain<AllDomain<T>>,
            VectorDomain<IntervalDomain<T>>, M, M>>
    where M: Metric,
          T: 'static + Clone + PartialOrd,
          M::Distance: DistanceConstant + One {
    if lower > upper {
      return fallible!(MakeTransformation,
        "lower may not be greater than upper")
    }
    Ok(Transformation::new(
        VectorDomain::new_all(),
        VectorDomain::new(IntervalDomain::new(
          Bound::Included(lower.clone()),
          Bound::Included(upper.clone()))),
        Function::new(
          move |arg: &Vec<T>|
          arg.iter().map(|e| clamp(&lower, &upper, e)).collect()),
        M::default(),
        M::default(),
        StabilityRelation::new_from_constant(M::Distance::one())))
}
```

Figure 1: Original Rust code for the vector clamp transformation.

```python
def MakeClamp(L: T, U: T):
    input_domain = VectorDomain(AllDomain(T))
    output_domain = VectorDomain(IntervalDomain(L, U))
    input_metric = SymmetricDistance()
    output_metric = SymmetricDistance()

    def relation(d_in: u32, d_out: u32) -> bool:
        return d_out >= d_in*1

    def function(data: Vec[T]) -> Vec[T]:
        def clamp(x: T) -> T:
            return max(min(x, U), L)
        return list(map(clamp, data))

    return Transformation(input_domain, output_domain,
    function, input_metric, output_metric,
    stability_relation = relation}
```

Figure 2: Python-like pseudocode for the vector clamp transformation; note the semantic distinctions from the Rust code shown in Figure 1.

Our aim is to replace the informal pseudocode used in the whitepapers proofs

with a Lean specification (examples of these specifications are shown in section 3). Our current approach requires an informal translation step, but reasoning about the translation is made precise within Lean. This approach allows for a gradual phasing out of the manual translation step—either by generating the Lean from Rust in a semi-automated fashion via shallow embedding as in [Ull16], by generating Rust from the Lean with a purpose-built transpiler, or calling Lean directly using FFI. Lean also enables the verification of correctness properties beyond those specified in Definition 1.4.

2.2 Container data types in Lean

Lean's standard library (augmented with mathlib) supports reasoning about relations between vectors, lists, and multisets. Reasoning about stability involves all of these objects: most transformations in take a vector as input, but we define symmetric distance in terms of multisets. It is possible to formulate a stricter definition of symmetric distance based on the change at each index over a pair of vectors; such a definition can be thought of as a binarized variant of the L_1 distance. However, the aggregation transformations we consider in this project (two variants of bounded sum) are permutation-invariant, so such a definition would be unnecessarily strict. Thus, our Lean model of OpenDP semantics uses the multiset definition of symmetric distance for transformations with a `SizedDomain` (vector) input domain.

In Lean's standard library, lists are defined according to the classic Lisp-like construction: a list is a polymorphic *inductive data type* with a `nil` constructor (which is used to construct an empty list) and a `cons` constructor (which is used to recursively construct a nonempty list that terminates with `nil`). Lists are parameterized by element type (that is, the type `list` $\mathbb{N}$ is distinct from the type `list` $\mathbb{R}$) and can only contain elements of this type.

Lean supports *dependent types*. This allows for the construction of types that carry highly specific bounds and other auxiliary information—for instance, the type `vector` $\mathbb{N}$ `10` denotes a vector with element type $\mathbb{N}$ and length 10 [Baa+20, p. 43]. Vectors in Lean are constructed as a subtype of lists with a length constraint [Baa+20, p. 177], yielding a two-line definition:

```
def vector (α : Type) (n : N) : Type
:= {xs : list α // list.length xs = n}
```

Lean also supports *quotient types*, which allow for the construction of new types from existing ones by introducing alternate equivalence relations. Multisets are defined in Lean as "as the quotient type over lists up to reordering" [Baa+20, p. 191]. Because `vectors` and `multisets` are both based on `lists`, it is possible to coerce data between the three representations (with possible loss of permutation and length information). This rich type system is a key distinction between the actual implementation of OpenDP in Rust and our model of OpenDP in Lean. In the Rust implementation, dependent types are simulated by wrapping type information in structures. For instance, the `SizedDomain`

structure carries equivalent information to the `vector` dependent type in Lean. Encoding constraints at the type level shifts the burden of verifying constraints to Lean's type checker, rather than the programmer, which makes it easier to write straightforwardly correct programs.

2.3 Machine arithmetic models in Lean

Dependent types can also be used to model machine arithmetic in Lean. Lean supports reasoning over $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{Z}$, and $\mathbb{N}$ (as well as other number systems, such as $\mathbb{C}$ and the p-adics), as well as restrictions on these systems. The `fin` type is a dependent type based on $\mathbb{N}$ with the addition of an upper bound constraint. It is defined in the standard library in one line:

```
def fin (n : N) := {i : N // i < n}
```

The standard library also defines basic arithmetic operations for `fin`, defined with modular arithmetic such that `fin` n and its associated operations resemble the ring $\mathbb{Z}/n\mathbb{Z}$. Thus, arithmetic defined on `fin 4294967296` (for instance) corresponds to unsigned 32-bit machine arithmetic. It would be relatively simple to model *signed* machine integers of arbitrary size by defining

```
def machine_int (lb ub : ℤ) := {i : ℤ // i > lb ∧ i < ub}
```

However, this would have required us to reconstruct the library of useful theorems already available for `fin` for this new type. Lean also includes a library for reasoning about finite-precision floating-point arithmetic, but it is relatively immature. Thus, we limit our analysis to the unsigned model.

3 Results

Here, we present selected snippets of Lean code used to verify OpenDP transformations. Full proofs are available at `github.com/pjrule/cs208-project`.

3.1 Distances and differences

Below, we show our construction of the symmetric distance metric on vectors (see Definition 1.3) using the hierarchy of container data types described in section 2.2.

```
variables {n m : ℕ} {α β: Type*} [decidable_eq α] [decidable_eq β] (f : α → β)

@[simp] def multiset.sym_diff (x y : multiset α) : multiset α
  := (x - y) + (y - x)
infix ` Δ `:51 := multiset.sym_diff

@[simp] def multiset.sym_dist (x y : multiset α) : ℕ := (x Δ y).card
infix ` |Δ| `:51 := multiset.sym_dist
```

```
@[simp] def list.sym_dist (x y : list α) : ℕ
  := (x : multiset α) |Δ| (y : multiset α)
infix ` |ΔL| `:52 := list.sym_dist

@[simp] def vector.sym_dist (x y : vector α n)
  := (x.to_list : multiset α) |Δ| (y.to_list : multiset α)
infix ` |ΔV| `:52 := vector.sym_dist

@[simp] def vec_neighbors (x y : vector α n) : bool := (x |ΔV| y) ≤ 1
infix ` ~ `:53 := vec_neighbors
```

We note a few characteristics typical to Lean code: the vector element types α and β are generic but constrained (using `decidable_eq`) such that equality between members of each type is decidable—a condition that is satisfied by the finite integer types we are interested in. This constraint ensures that these functions are *computable*—that is, we can both reason about them and execute them in the Lean interpreter. Using `infix`, we build up a convenient notation for referring to distances—for instance, `x` $|\Delta V|$ `y` refers to the symmetric distance between two vectors `x` and `y`. The `@[simp]` annotation instructs Lean's simplification tactic to use the definitions automatically.

3.2 Row transformations

Using the symmetric distance metric, we can reason about *row transformations*—that is, transformations where a function is applied individually to each element in a vector or multiset. In general, the number of elements that differ between two vectors or multisets after applying such a transformation is at most the number of elements that differed before the transformation (see the existing row transform proof in whitepapers). Because Lean does not include a notion of multiset symmetric difference (we constructed it in section 3.1), we must first show that this property holds for the one-sided difference by proving

```
theorem multiset.map_diff_subset (s₁ s₂ : multiset α) :
  s₁.map f - s₂.map f ≤ (s₁ - s₂).map f := (see code)
```

We then use `multiset.map_diff_subset`[3] to prove its symmetric analogue.

```
theorem multiset.map_sym_dist_le_sym_dist (s₁ s₂ : multiset α) :
  (s₁.map f) |Δ| (s₂.map f) ≤ s₁ |Δ| s₂ := (see code)
```

[3]Proving this theorem as a Lean novice turned out to be nontrivial. I worked on it for several hours before getting stuck and posting to the Lean community chat server, where Patrick Johnson posted a 160-line proof in about 14 hours (I am impressed by the responsiveness and friendliness of the experts in the Lean community). Junyan Xu later used a counting argument over multiset elements to dramatically simplify the proof to under 10 lines; this is the version of the proof I am using, with attribution, in my posted source code. In addition to Patrick and Junyan, I gratefully acknowledge Kevin Buzzard, Eric Wieser, and others for their contributions to this discussion.

Once these properties have been established, exhibiting a stability relation for arbitrary row transformations is trivial. Consider this definition of one-sided vector clamping based on the existing `fin.clamp` function. We formulate the definitions in terms of multisets for simplicity, but they can easily be translated to use vectors.

```
def clamp (ub : ℕ) (v : ℕ) : fin (ub + 1) := fin.clamp v ub

@[simp] def clamp_vec (s : multiset ℕ) (ub : ℕ) :
multiset (fin (ub + 1))
  := s.map (clamp ub)
```

Observe that the type of `clamp_vec` can be viewed as both a proof that the function's output domain is multisets with elements in $\mathbb{N} \cap [0, \texttt{ub}]$. Proving such a property (which corresponds to part (b) of Definition 1.4) about the equivalent Rust code using the pseudocode method is considerably more laborious. For the stability relation, we have

```
theorem clamp_stability (s₁ s₂ : multiset ℕ) (ub : ℕ) :
  (clamp_vec s₁ ub) |Δ| (clamp_vec s₂ ub) ≤ s₁ |Δ| s₂
:= multiset.map_sym_dist_le_sym_dist  _ _ _
```

That is, the desired stability relation follows immediately from the general `multiset.map_sym_dist_le_sym_dist` theorem. So part (c) of Definition 1.4 is satisfied. Furthermore, part (a) of Definition 1.4 (domain-metric compatibility) is satisfied, as the infix operator $|\Delta|$ is syntactic sugar for the `multiset.sym_dist` function—which only accepts multisets. Thus, `clamp_vec` satisfies all the conditions for a vetted transformation. Constructing such a proof for other row transformations is similarly trivial. For instance, here is a proof for an abbreviated Lean translation of the `is_equal` transformation for general multisets.

```
theorem is_equal_stability (s₁ s₂ : multiset α) (v : α) :
  (s₁.map (eq v)) |Δ| (s₂.map (eq v)) ≤ s₁ |Δ| s₂
:= multiset.map_sym_dist_le_sym_dist _ _ _
```

3.3 Bounded sum, known size

We use similar techniques to prove the correctness of a Lean translation of the bounded sum with unknown N transformation for integer types and exhibit a stability relation. Most of the difficulties in this proof relate to type coercion—we define the transformation in terms of vectors but use the multiset symmetric difference definition in the stability relation.

```
@[reducible] def vector.fin_sum (v : vector (fin (n + 1)) m) : ℕ
  := multiset.sum ((v.to_list : multiset (fin (n + 1))).map nat_of_fin)
```

We use dependent types to encode the input and output domains: a vector of length `m` and elements of type `fin (n + 1)` maps to a scalar of type `fin ((n * m) + 1`.

```
@[reducible] def vector.fin_bounded_sum (v : vector (fin (n + 1)) m) :
fin ((n * m) + 1)
  := fin.of_nat' (vector.fin_sum v)
```

We must demonstrate that our internal coercion from an unbounded natural number to a natural number bounded by mn is correct—that is, `fin.of_nat'` (which is essentially a modulo-nm operation) does not wrap around.

```
@[simp] theorem vector.fin_sum_le (v : vector (fin (n + 1)) m) :
  vector.fin_sum v ≤ n * m := (see code)
```

Once we have demonstrated these properties and defined the output metric, we can prove the main stability result, which has a similar form to the row transformation results.

```
@[simp] def vector.fin_bounded_sum_dist (v₁ v₂ : vector (fin (n + 1)) m) : ℕ
  := nat.dist (vector.fin_bounded_sum v₁) (vector.fin_bounded_sum v₂)

theorem vector.bounded_sized_sum_stability (v₁ v₂ : vector (fin (n + 1)) m) :
vector.fin_bounded_sum_dist v₁ v₂ ≤ n * (v₁ |ΔV| v₂) := (see code)
```

Once again, all parts of Definition 1.4 are satisfied by construction or by explicit proof.

4 Conclusion

We have demonstrated the feasibility of verifying basic OpenDP transformations in Lean. However, this work could be extended by

- *Proving the correctness of more—and more complicated—transformations.* We are close to achieving completeness parity, at least for unsigned integers, with the whitepapers project. However, it would be valuable to extend our efforts to more complicated transformations that involve more subtle typing issues, such as bounded variance. Work on a proof for the unsized bounded sum transformation is underway; it critically depends on the associativity of the saturating sum over numbers of the same sign, as discussed in [Wag22]
- *Reasoning about measurements.* For instance, it is possible to express the probability density function of the Laplace transformation in Lean via noncomputable functions. Is it possible to prove that a measurement yields the expected privacy guarantee in Lean?
- *Calling Lean from Rust or Python.* Is it possible to construct a high-assurance differential privacy library *purely* in Lean, albeit possibly at the cost of performance?

References

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*. Springer, 2002.

[Gon08] Georges Gonthier. "Formal proof–the four-color theorem". In: *Notices of the AMS* 55.11 (2008), pp. 1382–1393.

[Kle+09] Gerwin Klein et al. "seL4: Formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, pp. 207–220.

[RP10] Jason Reed and Benjamin C. Pierce. "Distance makes the types grow stronger: A calculus for differential privacy". In: *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. 2010, pp. 157–168.

[HPN11] Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. "Differential privacy under fire". In: *Proceedings of the 20th USENIX conference on Security*. SEC'11. USA: USENIX Association, Aug. 8, 2011, p. 33. (Visited on 04/06/2022).

[Mir12] Ilya Mironov. "On significance of the least significant bits for differential privacy". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. CCS '12. New York, NY, USA: Association for Computing Machinery, Oct. 16, 2012, pp. 650–661. ISBN: 978-1-4503-1651-4. DOI: 10.1145/2382196.2382264. URL: https://doi.org/10.1145/2382196.2382264 (visited on 04/06/2022).

[DR13] Cynthia Dwork and Aaron Roth. "The Algorithmic Foundations of Differential Privacy". In: *Foundations and Trends® in Theoretical Computer Science* 9.3 (2013), pp. 211–407. ISSN: 1551-305X, 1551-3068. DOI: 10.1561/0400000042. URL: http://www.nowpublishers.com/articles/foundations-and-trends-in-theoretical-computer-science/TCS-042 (visited on 04/07/2022).

[Gon+13] Georges Gonthier et al. "A machine-checked proof of the odd order theorem". In: *International conference on interactive theorem proving*. Springer, 2013, pp. 163–179.

[Bar+14] Gilles Barthe et al. "Proving Differential Privacy in Hoare Logic". In: 2014 IEEE 27th Computer Security Foundations Symposium (CSF). ISSN: 1063-6900. IEEE Computer Society, July 1, 2014, pp. 411–424. ISBN: 978-1-4799-4290-9. DOI: 10.1109/CSF.2014.36. URL: https://www.computer.org/csdl/proceedings-article/csf/2014/4290a411/12OmNz61dDR (visited on 05/03/2022).

[Mou+15] Leonardo de Moura et al. "The Lean theorem prover (system description)". In: *International Conference on Automated Deduction*. Springer, 2015, pp. 378–388.

[LSL16] Min Lyu, Dong Su, and Ninghui Li. "Understanding the Sparse Vector Technique for Differential Privacy". In: *arXiv:1603.01699 [cs]* (Sept. 17, 2016). arXiv: 1603.01699. URL: http://arxiv.org/abs/1603.01699 (visited on 04/07/2022).

[Ull16] Sebastian Ullrich. "Simple verification of rust programs via functional purification". In: *Master's Thesis, Karlsruher Institut für Technologie (KIT)* (2016).

[ZK17] Danfeng Zhang and Daniel Kifer. "LightDP: towards automating differential privacy proofs". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '17: The 44th Annual ACM SIGPLAN Symposium on Principles of Programming Languages. Paris France: ACM, Jan. 2017, pp. 888–901. ISBN: 978-1-4503-4660-3. DOI: 10.1145/3009837.3009884. URL: https://dl.acm.org/doi/10.1145/3009837.3009884 (visited on 04/07/2022).

[AH18] Aws Albarghouthi and Justin Hsu. "Synthesizing coupling proofs of differential privacy". In: *Proceedings of the ACM on Programming Languages* 2 (POPL Jan. 2018), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3158146. URL: https://dl.acm.org/doi/10.1145/3158146 (visited on 04/07/2022).

[JNS18] Noah Johnson, Joseph P. Near, and Dawn Song. "Towards Practical Differential Privacy for SQL Queries". In: *arXiv:1706.09479 [cs]* (Sept. 4, 2018). DOI: 10.1145/3177732.3177733. arXiv: 1706.09479. URL: http://arxiv.org/abs/1706.09479 (visited on 04/07/2022).

[BV19] Victor Balcer and Salil Vadhan. "Differential Privacy on Finite Computers". In: *arXiv:1709.05396 [cs]* (Jan. 17, 2019). arXiv: 1709.05396. URL: http://arxiv.org/abs/1709.05396 (visited on 04/07/2022).

[Mic19] Microsoft Research. *The Future of Mathematics?* Oct. 1, 2019. URL: https://www.youtube.com/watch?v=Dp-mQ3HxgDE (visited on 05/03/2022).

[Nea+19] Joseph P. Near et al. "Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy". In: *Proceedings of the ACM on Programming Languages* 3 (OOPSLA Oct. 10, 2019), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3360598. URL: https://dl.acm.org/doi/10.1145/3360598 (visited on 04/07/2022).

[Baa+20] Anne Baanen et al. *The Hitchhiker's Guide to Logical Verification*. 2020. URL: https://cs.brown.edu/courses/cs1951x/static_files/main.pdf.

[Joh+20] Noah Johnson et al. "Chorus: a Programming Framework for Building Scalable Differential Privacy Mechanisms". In: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)* (Sept. 2020), pp. 535–551. DOI: 10.1109/EuroSP48549.2020.00041. arXiv: 1809.07750. URL: http://arxiv.org/abs/1809.07750 (visited on 04/07/2022).

[Tea20] The OpenDP Team. *The OpenDP White Paper*. May 11, 2020. URL: https://projects.iq.harvard.edu/files/opendp/files/opendp_white_paper_11may2020.pdf.

[ADN21] Chike Abuah, David Darais, and Joseph P. Near. "Solo: A Lightweight Static Analysis for Differential Privacy". In: *arXiv:2105.01632 [cs]* (Oct. 13, 2021). arXiv: 2105.01632. URL: http://arxiv.org/abs/2105.01632 (visited on 04/07/2022).

[CTW21a] Sílvia Casacuberta, Grace Tian, and Connor Wagaman. *List of definitions used in proofs*. Sept. 8, 2021. URL: https://github.com/opendp/whitepapers/blob/f43bf7056a5fd3b6f7b4bb77d451eafa042fe8f7/proof-defns/proof_defns.pdf.

[CTW21b] Sílvia Casacuberta, Grace Tian, and Connor Wagaman. *List of definitions used in the pseudocode*. Sept. 23, 2021. URL: https://github.com/opendp/whitepapers/blob/cfab535367f592c1242ab880002cc726822506a0/pseudocode-defns/pseudocode_defns.pdf.

[21] *whitepapers*. original-date: 2021-07-06T21:04:09Z. Oct. 25, 2021. URL: https://github.com/opendp/whitepapers (visited on 05/01/2022).

[Goo22] Google. *Differential Privacy*. original-date: 2019-09-04T13:04:15Z. Apr. 5, 2022. URL: https://github.com/google/differential-privacy (visited on 04/07/2022).

[Pro22a] Harvard University Privacy Tools Project. *OpenDP*. original-date: 2021-02-06T17:40:39Z. Mar. 31, 2022. URL: https://github.com/opendp/opendp (visited on 04/07/2022).

[Pro22b] Harvard University Privacy Tools Project. *opendp/smartnoise-core*. original-date: 2019-09-04T19:46:55Z. Mar. 29, 2022. URL: https://github.com/opendp/smartnoise-core (visited on 04/07/2022).

[Wag22] Connor Wagaman. "Finite-Precision Arithmetic Isn't Real: The Impact of Finite Data Types on Efforts to Fulfill Differential Privacy on Computers". Cambridge, MA: Harvard University, Mar. 25, 2022.

[Ope] OpenDP. *Code Structure*. URL: https://docs.opendp.org/en/stable/developer/code-structure.html (visited on 05/03/2022).