LightDP: Towards Automating Differential Privacy Proofs

Danfeng Zhang Daniel Kifer

Department of Computer Science and Engineering
Penn State University
University Park, PA United States
{zhang,dkifer@cse.psu.edu}

Abstract

The growing popularity and adoption of differential privacy in academic and industrial settings has resulted in the development of increasingly sophisticated algorithms for releasing information while preserving privacy. Accompanying this phenomenon is the natural rise in the development and publication of incorrect algorithms, thus demonstrating the necessity of formal verification tools. However, existing formal methods for differential privacy face a dilemma: methods based on customized logics can verify sophisticated algorithms but come with a steep learning curve and significant annotation burden on the programmers, while existing programming platforms lack expressive power for some sophisticated algorithms.

In this paper, we present LightDP, a simple imperative language that strikes a better balance between expressive power and usability. The core of LightDP is a novel relational type system that separates relational reasoning from privacy budget calculations. With dependent types, the type system is powerful enough to verify sophisticated algorithms where the composition theorem falls short. In addition, the inference engine of LightDP infers most of the proof details, and even searches for the proof with minimal privacy cost when multiple proofs exist. We show that LightDP verifies sophisticated algorithms with little manual effort.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs.

Keywords Differential privacy; dependent types; type inference;

1. Introduction

Companies, government agencies, and academics are interested in analyzing and modeling datasets containing sensitive information about individuals (e.g., medical records, customer behavior, etc.). Privacy concerns can often be mitigated if the algorithms used to manipulate the data, answer queries, and build statistical models satisfy differential privacy (Dwork et al. 2006b) — a set of restrictions on their probabilistic behavior that provably limit the ability of attackers to infer individual-level sensitive information (Dwork et al. 2006b; Kifer and Machanavajjhala 2014).

Since 2006, differential privacy has seen explosive growth in many areas, including theoretical computer science, databases, machine learning, and statistics. This technology has been deployed in practice, starting with the U.S. Census Bureau LEHD OnTheMap tool (Machanavajjhala et al. 2008), the Google Chrome Browser (Erlingsson et al. 2014), and Apple's new data collection efforts (Greenberg 2016). However, the increase in popularity and usage of differential privacy has also been accompanied by a corresponding increase in the development and implementation of algorithms with flawed proofs of privacy; for example, Chen and Machanavajjhala (2015) and Lyu et al. (2016) catalog some recent cases about variations of the Sparse Vector method (Dwork and Roth 2014) .

Currently, there are two strategies for combating this trend. The first is the use of programming platforms (McSherry 2009; Mohan et al. 2012; Roy et al. 2010) that have privacy primitives that restrict the privacy-preserving algorithms that can be implemented and often add more noise than is necessary to the computation. The second strategy is the development of languages and formal verification tools for differential privacy (Reed and Pierce 2010; Gaboardi et al. 2013; Barthe et al. 2012, 2014, 2016c,b). These languages enable the development of much more sophisticated algorithms that use less noise and hence provide more accurate outputs. However, the increased power of the formal methods comes with a considerable cost — a programmer has to heavily annotate code and generate proofs using complicated logics such as a customized relational Hoare logic proposed by Barthe et al. (2012). Moreover, intricate proof details have to be provided by a programmer, which makes exploring variations of an algorithm difficult since small variations in code can cause significant changes to a proof.

In this paper, we present LightDP, a language for developing provably privacy-preserving algorithms. The goal of LightDP is to minimize the burden on the programmer while retaining most of the capabilities of the state-of-the-art, such as verifying the Sparse Vector method (Dwork and Roth 2014) (an algorithm which, until very recently (Barthe et al. 2016c,b), was beyond the capabilities of verification tools). For example, we show that the Sparse Vector method can be verified in LightDP with little manual effort: just two lines of annotation from the programmer.

LightDP is equipped with a novel light-weight relational type system that clearly separates relational reasoning from privacy budget calculation. In particular, it transforms the original probabilistic program into an equivalent nonprobabilistic program, where all privacy costs become explicit. With dependent types, the explicitly calculated privacy cost in the target language may depend on program states, hence enabling the verification of sophisticated algorithms (e.g., the Sparse Vector method) that are beyond the capability of many existing methods (Reed and Pierce 2010; Gaboardi et al. 2013; Barthe et al. 2012, 2014) based on the composition the-

POPL'17, January 15–21, 2017, Paris, France
© 2017 ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009884

orem (McSherry 2009). Moreover, the transformed nonprobabilistic program is ready for off-the-shelf formal verification methods, such as Hoare logic, to provide an upper bound of the privacy cost.

On the usability end, LightDP has an inference engine that reduces the already low annotation burden on the programmers. Although the inference engine does not yet automate the privacy budget calculation part of a proof, it does fill in missing details in the relational reasoning part of a proof; furthermore, based on MaxSMT theory, it even searches for the optimal proof that minimizes privacy cost with minimal human involvement. For example, with only one postcondition annotation and one loop invariant annotation from a programmer, LightDP confirms that the proof in (Dwork and Roth 2014) indeed provides the minimal privacy cost.

To summarize, this paper makes the following contributions:

1. LightDP, a new imperative language for verifying sophisticated privacy-preserving algorithms (Section 3.1),
2. expressive static annotations incorporating dependent types, enabling precise tracking of privacy costs (Section 3.3),
3. a formal proof that the LightDP type system soundly tracks differential privacy costs, and new proof techniques involved in the soundness proof (Section 4),
4. an inference engine that automatically fills in missing details involved in the relational reasoning part of a proof, and further, minimizes provable privacy cost when multiple proofs exist, with little manual effort (Section 5),
5. case studies on complex algorithms showing that formal verification of privacy-preserving algorithms are viable with little programmer annotation burden (Section 6).

2. Preliminaries and Illustrating Example

2.1 Distributions

We define the set of *sub-distributions* over a set A, written $\mathbf{Dist}(A)$, as the set of functions $\mu : A \to [0,1]$, such that $\sum_{a\in A} \mu\ a \leq 1$. When applied to an event $E \subseteq A$, we define $\mu(E) \triangleq \sum_{e\in E} \mu(e)$. Notice that we do not require $\sum_{a\in A} \mu\ a = 1$, a special case when μ is a *distribution*, since sub-distribution gives rise to an elegant semantics for programs that may not terminate (Kozen 1981).

Given a distribution $\mu \in \mathbf{Dist}(A)$, its support is defined as $\texttt{support}(\mu) \triangleq \{a \mid \mu(a) > 0\}$. We use $\mathbb{1}_a$ to represent the degenerate distribution μ that $\mu(a) = 1$ and $\mu(a') = 0$ if $a' \neq a$. Moreover, sub-distributions can be given a structure of a monad. Formally, we define the `unit` and `bind` functions as follows:

$$\texttt{unit} : A \to \mathbf{Dist}(A) \triangleq \lambda a.\ \mathbb{1}_a$$
$$\texttt{bind} : \mathbf{Dist}(A) \to (A \to \mathbf{Dist}(B)) \to \mathbf{Dist}(B)$$
$$\triangleq \lambda\mu.\ \lambda f.\ (\lambda b.\ \sum_{a\in A} (f\ a\ b) \times \mu(a))$$

That is, `unit` takes an element in A and returns the Dirac distribution where all mass is assigned to a; `bind` takes μ, a distribution on A, and f, a mapping from A to distributions on B (e.g., a conditional distribution of B given A), and returns the corresponding marginal distribution on B. This monadic view will avoid cluttered definitions and proofs when probabilistic programs are involved (Section 3.2).

2.2 Differential Privacy

Differential privacy has two major variants: *pure* (Dwork et al. 2006b) (obtained by setting $\delta = 0$ in the following definition) and *approximate* (Dwork et al. 2006a) (obtained by choosing a $\delta > 0$).

Definition 1 (Differential privacy)**.** *Let $\epsilon, \delta \geq 0$. A probabilistic computation $M : A \to \mathbf{Dist}(B)$ is (ϵ, δ)-differentially private with respect to an adjacency relation $\Psi \subseteq A \times A$ if for every pair of inputs $a_1, a_2 \in A$ such that $a_1 \Psi a_2$, and every output subset $E \subseteq B$, we have*

$$P(Ma_1 \in E) \leq \exp(\epsilon) P(Ma_2 \in E) + \delta$$

Intuitively, a probabilistic computation satisfies differential privacy if it produces similar distributions for any pair of inputs related by Ψ. In the most common applications of differential privacy, A is the set of possible databases and the adjacency relation Ψ is chosen so that $a_1 \Psi a_2$ whenever a_1 can be obtained from a_2 by adding or removing data belonging to a single individual.

In this paper, we focus on the verification of algorithms that satisfy pure differential privacy. An algorithm is ϵ-differentially private iff it is $(\epsilon, 0)$-differentially private according to Definition 1.

function SPARSEVECTOR (T, N, ϵ : $\texttt{num}_0$; q : $\texttt{list num}_*$)
returns (out : $\texttt{list bool}_0$)
precondition $\forall i.\ -1 \leq (\widehat{q}[i]) \leq 1$

$\texttt{c1}, \texttt{c2}, \texttt{i} : \texttt{num}_0; \tilde{T}, \eta_1 : \texttt{num}_1; \eta_2 : \texttt{num}_{q[i]+\eta_2 \geq \tilde{T} ? 2:0}$

```
1   η1 := Lap (2/ε);
2   T̃ := T + η1;
3   c1 := 0; c2 := 0; i := 0;
4   while (c1 < N)
5     η2 := Lap (4N/ε);
6     if (q[i] + η2 ≥ T̃) then
7       out:= true::out;
8       c1 := c1 + 1;
9     else
10      out:= false::out;
11      c2 := c2 + 1;
12    i := i+1;
```

Figure 1. The Sparse Vector method. Type annotations are shown in grey. The precondition specifies the adjacency assumption.

2.3 The Sparse Vector Method

The goal of formal methods for verifying ϵ-differential privacy is to provide an upper bound on the privacy cost ϵ of a program. Typically, users will have a fixed privacy budget ϵ' and can only run programs whose provable privacy cost ϵ does not exceed the budget: $\epsilon \leq \epsilon'$. For this reason, it is important that formal methods are able to prove a tight upper bound on the privacy cost.

With the exception of (Barthe et al. 2016c,b), most existing formal methods rely on the composition theorem (McSherry 2009). That is, in the case of ϵ-differential privacy, those methods essentially treat a program as a series of modules, each with a provable upper bound ϵ_i on its privacy cost. Then by the composition theorem (McSherry 2009), the total privacy cost is bounded by $\sum_i \epsilon_i$. However, for sophisticated advanced algorithms, the composition theorem often falls short — it can provide upper bounds that are arbitrarily larger than the true privacy cost. Providing the tightest privacy cost for intricate algorithms requires formal methods that are more powerful but avoid over-burdening the programmers with annotation requirements. To illustrate these challenges, we consider the Sparse Vector method (Dwork and Roth 2014). It is a prime example of the need for formal methods because many of its published variants have been shown to be incorrect (Chen and Machanavajjhala 2015; Lyu et al. 2016).

The Sparse Vector method also has many correct variants, one of which is shown in Figure 1. For now, safely ignore the type annotations in grey and the precondition. Here, the input list q represents a sequence of results of counting queries $q_1, q_2, q_3, \ldots$ (e.g., how many patients in the data have cancer, how many patients con-

tracted an infection in the hospital, etc.) running on a database. The goal is to answer as accurately as possible the following question: which queries, when evaluated on the true database, return an answer greater than the threshold T (a program input unrelated to the sensitive data)?

To achieve differential privacy, the algorithm adds appropriate Laplace noise to the threshold and to each query. Here, `Lap` $(4N/\epsilon)$ draws one sample from the Laplace distribution with mean zero and a scale factor $(4N/\epsilon)$. If the noisy query answer $(q[i] + \eta_2)$ is above the noisy threshold $\tilde{T}$, it adds `true` to the output list `out` (in the slot reserved for that query) and otherwise, it adds `false`. The key to this algorithm is the deep observation that once noise has been added to the threshold, queries for which we output `true` have a privacy cost (so we can answer at most N of them, where N is a parameter); however, outputting `false` for a query does not introduce any new privacy costs (Dwork and Roth 2014). The algorithm ensures that the total privacy cost is bounded by the input ϵ, the parameter used in Figure 1. This remarkable property makes the Sparse Vector method ideal in situations where the vast majority of query counts are expected to be below the threshold.

Failure of the composition theorem If we just use the composition theorem, we would have a privacy cost of $\epsilon/4N$ for each loop iteration (i.e., every time `Lap` $(4N/\epsilon)$ noise is added to a query answer) due to the property of the Laplace distribution. Since the number of iterations are not a priori bounded, the composition theorem could not prove that the algorithm satisfies ϵ'-differential privacy for any finite ϵ'; more advanced methods are needed.

Informal proof and sample runthrough Proofs of correctness (of this and other variants) can be found in (Dwork and Roth 2014; Chen and Machanavajjhala 2015; Lyu et al. 2016). Here we provide an informal correctness argument by example to illustrate the subtleties involved both in proving it and inferring a tight bound for the algorithm.

Suppose we set the parameters $T = 4$ (we want to know which queries have a value at least 4) and $N = 1$ (we stop the algorithm after the first time it outputs `true`). Consider the following two databases D_1, D_2 that differ on one record, and their corresponding query answers:

$$D_1: \quad q[0] = 2, \quad q[1] = 3, \quad q[2] = 5$$
$$D_2: \quad q[0] = 3, \quad q[1] = 3, \quad q[2] = 4$$

Suppose in one execution on D_1, the noise added to T is $\alpha^{(1)} = 1$ and the noise added to $q[0], q[1], q[2]$ is $\beta_0^{(1)} = 2, \beta_1^{(1)} = 0, \beta_2^{(1)} = 0$, respectively. Thus the noisy threshold is $\tilde{T} = 5$ and the noisy query answers are $q[0]+\beta_0^{(1)} = 4, q[1]+\beta_1^{(1)} = 3, q[2]+\beta_2^{(1)} = 5$ and so the algorithm outputs the sequence: (`false`, `false`, `true`).

According to Definition 1, for any output sequence ω, we need to show $P(M(D_1) = \omega) \leq e^{\epsilon} P(M(D_2) = \omega)$ for all possible outputs ω and databases D_1, D_2 that differ on one record. For the databases D_1, D_2 described above, we will show that $P(M(D_1) = (\texttt{false}, \texttt{false}, \texttt{true})) \leq e^{\epsilon} P(M(D_2) = (\texttt{false}, \texttt{false}, \texttt{true}))$. We proceed in two steps.

Aligning randomness We first create an *injective* (but not necessarily bijective) function from the randomness in the execution under D_1 into the randomness in the execution under D_2, so that both executions generate the same output. For an execution under D_2, let $\alpha^{(2)}$ be the noise added to the threshold and let $\beta_0^{(2)}, \beta_1^{(2)}, \beta_2^{(2)}$ be the noise added to the queries $q[0], q[1], q[2]$, respectively. Consider an injective function candidate that adds 1 to the threshold noise (i.e., $\alpha^{(2)} = \alpha^{(1)} + 1$), keeps the noise of queries for which D_1 reported `false` (i.e., $\beta_0^{(2)} = \beta_0^{(1)}$ and $\beta_1^{(2)} = \beta_1^{(1)}$) and adds 2 to the noise of queries for which D_1 reported `true` (i.e., $\beta_2^{(2)} = \beta_2^{(1)} + 2$).

In our running example, execution under D_2 with this function would result in the noisy threshold $\tilde{T} = 6$ and noise query answers $q[0] + \beta_0^{(2)} = 5$, $q[1] + \beta_1^{(2)} = 3$, $q[2] + \beta_2^{(2)} = 6$. Hence, the output once again is (`false`, `false`, `true`). In fact, it is easy to see that under this injective function, every execution under D_1 would result in an execution under D_2 that produces the same answer.

Counting privacy cost For each output ω, let f be the injective function; let A be the set of random variable assignments that cause execution under D_1 to produce ω; let B be the possible assignments we can get by applying f to A; and let C be the set of random variable assignments that are not in the range of f, but nevertheless cause execution under D_2 to produce the output ω as well[1]. Then we can rewrite $P(M(D_1) = \omega) = P(A)$ and $P(M(D_2) = \omega) = P(B) + P(C)$.

Once we have done the alignment of randomness as above, and recalling that $N = 1$ in our example, the proof finishes by showing:

$$P(A) = \sum_{a \in A} P(a) \leq e^{\frac{\epsilon}{2}} e^{2\frac{\epsilon}{4N}} \sum_{a \in A} P(f(a))$$
$$= e^{\epsilon} P(B) \leq e^{\epsilon}(P(B) + P(C))$$

where the $e^{\frac{\epsilon}{2}}$ factor results from using a threshold value that is 1 larger, while the $e^{2\frac{\epsilon}{4N}}$ factor results from adding 2 to the noise for query $q[2]$. Notice that no privacy cost is paid for queries $q[0]$ and $q[1]$, since the same noise is added under D_1 and D_2. Moreover, due to the injective assumption, $\sum_{a \in A} P(f(a)) = P(B)$.

Challenges The Sparse Vector method is a prime example of the need for formal methods, since paper-and-pencil proof is shown to be error-prone for its variants. The intricacy in its proof brings major challenges for formal methods:

1. Precision: a crucial observation in the proof is that once noise has been added to the threshold, different privacy costs are only paid for outputting `true`. Hence, the cost calculation needs to consider program states.
2. Aligning randomness: finding an injective function from the randomness under D_1 to that under D_2 such that outputting ω under D_1 entails outputting ω under D_2 is the most intriguing piece in the proof. However, coming up with a correct function, as we described informally above, is non-trivial.
3. Finding the tightest bound: in fact, an infinite number of proofs exist for the Sparse Vector method, though with various provable privacy costs[2]. Since a tighter privacy cost bound allows a privacy-preserving algorithm to produce more accurate outputs, a formal method should produce the tightest one when possible.

Except the very recent work by Barthe et al. (2016c,b), existing formal methods (e.g., (Reed and Pierce 2010; Gaboardi et al. 2013; Barthe et al. 2012, 2014)) rely on the composition theorem, hence fail to prove that the Sparse Vector method satisfies ϵ'-privacy for any ϵ'. Recent work by Barthe et al. (2016c,b) verifies variants of the Sparse Vector method, but its customized relational logic incurs heavy annotation burden, including the randomness alignment. Moreover, those work cannot search for the tightest cost bound.

[1] This is possible since we do not assume the function to be a bijection.

[2] For example, another injective function adds 2 to the threshold noise (i.e., $\alpha^{(2)} = \alpha^{(1)} + 2$), keeps the noise of queries for which D_1 reported `false` (i.e., $\beta_0^{(2)} = \beta_0^{(1)}$ and $\beta_1^{(2)} = \beta_1^{(1)}$) and adds 3 to the noise of queries for which D_1 reported `true` (i.e., $\beta_2^{(2)} = \beta_2^{(1)} + 3$). It is easy to check that this mapping has the desired property, but the privacy cost is $(7\epsilon)/4$.

2.4 Our Approach

To tackle the challenges above, we propose LightDP, an imperative language that enables verification and even inference of the tightest privacy cost for sophisticated privacy-preserving algorithms. We illustrate the key components of LightDP in this section, and detail all components in the rest of this paper.

Relational reasoning The core of LightDP is a novel light-weight dependent type system that explicitly captures the *exact difference* of a variable's values in two executions under two adjacent databases. Let $v^{(1)}$ ($v^{(2)}$) be the value of a variable x in an execution under D_1 (D_2). The type τ for x in LightDP has the form of $\mathcal{B}_{\mathbb{d}}$, meaning that x holds a value of basic type $\mathcal{B}$ (e.g., `int`, `real`, `bool`), and $v^{(1)} + \mathbb{d} = v^{(2)}$. Required type annotations for the Sparse Vector method is shown in grey in Figure 1. Note that for brevity, we write `num` for numeric base types (e.g., `int`, `real`). Hereafter, we refer to the $\mathbb{d}$ counterpart as the *distance*.

In the simplest case, the distance is a constant. For example, the input threshold T has a type $\texttt{num}_0$, meaning that its value remains the same in two executions (since T is a parameter unrelated to private information about individuals). The distance of variable η_1 captures one randomness alignment in the informal proof above: we enforce a distance of 1 for η_1 (i.e., we map noise v to $v+1$ for any value v sampled in the execution under D_1).

The distance may also depend on program states. Hence, LightDP supports dependent types. For instance, consider the distance of η_2: $q[i] + \eta_2 \geq \tilde{T}?2 : 0$. This annotation specifies an injective function that maps noise v to $v+2$ when the value of $q[i] + \eta_2 \geq \tilde{T}$ is `true` (i.e., the output is `true`) in an execution under D_1, and maps noise v to v otherwise. As we will see shortly, dependent types allow a precise privacy cost to be calculated under various program states, hence enabling bounding privacy cost in a tighter way than mechanisms based on the composition theorem.

Moreover, LightDP uses a distinguished distance $*$ as a shorthand for the standard Sigma type (e.g., $\texttt{num}_* \triangleq \Sigma_{x:\texttt{num}_0}\ \texttt{num}_x$). In other words, each value of type $\texttt{num}_*$ (e.g., each query in the list q) can be interpreted as a pair of the form $(x : \texttt{num}_0, y : \texttt{num}_x)$, where the first component specifies the distance of the second component. Note that by language design, the first component is invisible in the privacy-preserving algorithm; however, the type system may reason about and manipulate it via a distinguished operation $\widehat{\ }$. For instance, the precondition in the running example states the adjacent assumption on databases: for each query answer $q[i]$ in q, its distance $(\widehat{q}[i])$ is bounded by ± 1. The star type is also useful for distances that cannot be easily captured at compile time (Section 3.1).

With type annotations, a type system statically verifies that the distances are maintained as an invariant throughout the execution. For example, the output `out` in Figure 1 has type $\texttt{list bool}_0$, meaning that each element in the list has type $\texttt{bool}_0$. Hence, the invariant maintained by the type system ensures that two related executions always generate the same output.

Calculating privacy cost When type-checking succeeds, the type system transforms the original program to a non-probabilistic, non-relational program where privacy cost is explicitly calculated. The transformed program for the Sparse Vector method is shown in Figure 2.

The transformed program is almost identical to the original one, except that: 1) the privacy cost is explicitly calculated via a new variable $\mathbf{v}_\epsilon$, and 2) probabilistic instructions are replaced by a new nondeterministic instruction `havoc` η, which semantically sets variable η to an arbitrary value upon execution.

The fundamental soundness theorem of the type system states that, informally, if 1) the original program type-checks, and 2) $\mathbf{v}_\epsilon$ is always bounded by some constant ϵ' in the transformed program, then the original program being verified is ϵ'-differentially private.

function TSPARSEVEC (T, N, ϵ : num; q : list num; $\widehat{q}$: list num)
returns (out : list bool)
precondition $\forall i.\ -1 \leq (\widehat{q}[i]) \leq 1$

```
1   v_ε := 0;
2   havoc η1; v_ε := v_ε + ε/2;
3   T̃ := T + η1;
4   c1 := 0; c2 := 0; i := 0;
5   while (c1 < N)
      Invariant : c1 ≤ N ∧ v_ε = ε/2 + c1 × ε/2N
6     havoc η2; v_ε := v_ε + (q[i] + η2 ≥ T̃?2 : 0) × ε/4N;
7     if (q[i] + η2 ≥ T̃) then
8       out := true::out;
9       c1 := c1+1;
10    else
11      out := false::out;
12      c2 := c2+1;
13    i := i+1;
```

Figure 2. The transformed program. The instrumented statements are underlined. The loop invariant to prove the postcondition $\mathbf{v}_\epsilon \leq \epsilon$ is shown in grey.

Notice that for the second property (the problem of bounding $\mathbf{v}_\epsilon$), any off-the-shelf verification tool for functional correctness can be utilized. For instance, the program above with the desired postcondition $\mathbf{v}_\epsilon \leq \epsilon$ can be verified by Hoare logic, with one loop invariant provided by a programmer (the grey box in Figure 2).

2.5 Type Inference

The mechanisms sketched so far provide a light-weight yet powerful formal method for differential privacy. The annotation burden is much reduced compared with (Barthe et al. 2016c). However, providing the *correct* and *optimal* type annotations (especially for random variables η_1 and η_2) is still subtle for a programmer.

Although it is folklore that type inference in face of dependent types can be daunting, LightDP is equipped with an inference engine that, at least for many algorithms, not only infers correct annotations, but also enables finding annotations that minimize privacy cost when multiple annotations exist. For example, given the function signature in Figure 1, the inference algorithm in Section 5 automatically infers types for all variables. The inferred types are identical to the ones in Figure 1 except that $\tilde{T}$, η_1 are assigned with a distance α and η_2 is assigned with a distance expression $q[i]+\eta_2 \geq \tilde{T}?\beta : \gamma$, where α, β, γ are variables to be inferred, subject to constraints generated during type checking. For the Sparse Vector method, multiple correct type annotations exist. For example, $\alpha = 1, \beta = 2, \gamma = 0$ corresponds to the annotation in Figure 1. Moreover, $\alpha = 0, \beta = 2, \gamma = -2$ and $\alpha = 2, \beta = 3, \gamma = 0$ are both correct annotations.

Given type annotations with distance variables, the type-guided transformation as sketched above generates target program where variables to be inferred (i.e., α, β, γ) are used in the calculation of privacy cost. The difference is that $\mathbf{v}_\epsilon$ increments by $(\alpha\epsilon)/2$ at line 2 and increments by $(q[i] + \eta_2 \geq \tilde{T}?\beta : \gamma) \times \epsilon/4N$ at line 6 in Figure 2. By Hoare logic, we can easily bound the privacy cost to be $\alpha\epsilon/2 + \beta\epsilon/4 + \texttt{c2} \times \gamma\epsilon/4N$.

Putting it all together, finding the optimal proof is equivalent to a MaxSMT problem: $\min(2\alpha + \beta + \texttt{c2} \times \gamma/N)$, given that constraints generated in type checking are satisfiable. Using an existing MaxSMT solver, μZ (Bjørner and Phan 2014; Bjørner et al. 2015), the optimal proof for the Sparse Vector method is successfully inferred: $\alpha = 1, \beta = 2, \gamma = 0$. This is exactly the randomness alignment used in its proof (Dwork and Roth 2014).

Reals	r	$\in$	$\mathbb{R}$
Booleans	b	$\in$	$\{\texttt{true}, \texttt{false}\}$
Vars	x	$\in$	*Var*
Rand Vars	η	$\in$	H
Linear Ops	$\oplus$	::=	$+ \mid -$
Other Ops	$\otimes$	::=	$\times \mid /$
Comparators	$\odot$	::=	$< \mid > \mid = \mid \leq \mid \geq$
Rand Exps	g	::=	$\texttt{Lap}\ r$
Expressions	e	::=	$r \mid b \mid x \mid \eta \mid e_1 \oplus e_2 \mid e_1 \otimes e_2 \mid e_1 \odot e_2 \mid \neg e \mid e_1 :: e_2 \mid e_1[e_2] \mid e_1?e_2 : e_3$
Commands	c	::=	$\texttt{skip} \mid x := e \mid \eta := g \mid c_1; c_2 \mid \texttt{return}\ e \mid \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c$
Distances	$\mathbb{d}$	::=	$r \mid x \mid \eta \mid \mathbb{d}_1 \oplus \mathbb{d}_2 \mid \mathbb{d}_1 \otimes \mathbb{d}_2 \mid \mathbb{d}_1 \odot \mathbb{d}_2?\mathbb{d}_3 : \mathbb{d}_4$
Types	τ	::=	$\texttt{num}_{\mathbb{d}} \mid \texttt{num}_* \mid \texttt{bool} \mid \texttt{list}\ \tau \mid \tau_1 \to \tau_2$

Figure 3. LightDP$_0$: language syntax.

3. LightDP: A Language for Algorithm Design

We first introduce a simple imperative language, LightDP$_0$, for designing and verifying privacy-preserving algorithms. This language is equipped with a dependent type system that enables formal verification of sophisticated algorithms where the composition theorem falls short. In this section, we assume all type annotations are provided by a programmer. We will remove this restriction and enable type inference in Section 5.

3.1 Syntax

The language syntax is given in Figure 3. LightDP$_0$ is mostly a standard imperative language except for the following features.

Random expressions Probabilistic reasoning is essential in privacy-preserving algorithms. We use g to represent a random expression. Since LightDP$_0$ follows a modular design where new randomness expression can be added easily, we only consider the most interesting random expression, $\texttt{Lap}\ r$, for now. Semantically, $\texttt{Lap}\ r$ draws one sample from the Laplace distribution, with mean zero and a scale factor r. We will discuss other random expressions in Section 6.3.

Each random expression g can be assigned to a random variable η, written as $\eta := g$. We distinguish random variables (H) from normal variables (*Var*) for technical reasons explained in Section 3.3. Notice that although the syntax restricts the distribution scale parameters to be a constant, its mean can be an arbitrary expression e, via the legit expression $e + \eta$, where η is sampled from a distribution with mean zero.

List operations Sophisticated algorithms usually make multiple queries to a database and produce multiple outputs during that process. Rather than reasoning about the privacy cost associated with each query in isolation and total the privacy costs using the composition theorem, LightDP$_0$ enables more precise reasoning via built-in list type operations: $e_1 :: e_2$ appends the element e_1 to a list e_2; $e_1[e_2]$ gets the e_2-th element in list e_1, assuming e_2 is bound by the length of e_1. We also assume a list variable is initialized to an empty list.

Types with distances Each type τ has the form of $\mathcal{B}_{\mathbb{d}}$. Here, $\mathcal{B}$ is a base type, such as `num` (numeric type), `bool` (Boolean), or an application of a type constructor (e.g., `list`) to another type, or a function ($\tau_1 \to \tau_2$). $\mathbb{d}$ is a numeric expression that (semantically) specifies the exact distance of the values stored in a variable in two related executions. In particular, a distance expression is a numeric expression in the language, as specified in Figure 3, where $\mathbb{d}_1 \odot \mathbb{d}_2?\mathbb{d}_3 : \mathbb{d}_4$ evaluates to $\mathbb{d}_3$ when the comparison evaluates to `true`, and $\mathbb{d}_4$ otherwise.

function PARTIALSUM ($\epsilon, b, \texttt{size} : \texttt{num}_0; q : \texttt{list num}_*$)
returns ($\texttt{out} : \texttt{num}_0$)
precondition $\forall i \geq 0.\ \widehat{q}[i] \leq b \wedge$
$\forall i \geq 0.\ \widehat{q}[i] > 0 \Rightarrow (\forall j > i.\ \widehat{q}[j] = 0)$

```
1  sum : num*; i : num0; η : num_{-ŝum}
2  sum := 0; i := 0;
3  while (i < size)
4    sum := sum+q[i];
5    i := i+1;
6  η = Lap b/ε;
7  out := sum + η;
```

Figure 4. An ϵ-differentially private algorithm for summing over a query list.

Since non-numeric types `bool`, $\texttt{list}\ \tau$ and $\tau_1 \to \tau_2$ cannot be associated with any numeric distance, those types are syntactic sugars for $\texttt{bool}_0$, $(\texttt{list}\ \tau)_0$ and $(\tau_1 \to \tau_2)_0$ respectively. Notice that elements in a list of type $(\texttt{list}\ \tau)_0$ (e.g., parameter q in Figure 1) may still have different elements in two related executions, since the difference of elements is specified by type τ. The subscript 0 here simply restricts the list size in two related executions.

Star type LightDP$_0$ also supports sum types, written as $\mathcal{B}_*$, a syntactic sugar of a Sigma type. More specifically, a variable x with type $\texttt{num}_*$ is desugared as $x : \Sigma_{(\widehat{x}:\texttt{num}_0)}\ \texttt{num}_{\widehat{x}}$, where $\widehat{x}$ is a distinguished variable invisible in the source code, but can be reasoned about and manipulated by the type system. Hiding the first component of a Sigma type simplifies verification (Section 4).

The parameter q in Figure 1 is one example where the star type is useful. Moreover, the star type enables reasoning about dependencies that cannot be captured otherwise by a distance expression. Consider the ϵ-differentially private Partial Sum algorithm in Figure 4. It implements an immediate solution to answering the sum of a query list in a privacy preserving manner[3]: it aggregates the accurate partial sum in a loop, and releases a noisy sum using the Laplace mechanism. The precondition specifies the adjacency assumption: at most one query answer may differ by at most b.

In this algorithm, the distance of variable `sum` changes in each iteration. Hence, the accurate type for `sum` is $\texttt{num}_{\sum_{j=0}^{i} \widehat{q}[j]}$. However, with the goal of keeping type system as light-weight as possible, we assign `sum` to a star type. The type system will reason about and manipulate distance component $\widehat{\texttt{sum}}$ in a sound way (Section 3.3).

3.2 Semantics

The denotational semantics of the probabilistic language is defined as a mapping from initial memory to a distribution on (possible) final outputs. Formally, let $\mathcal{M}$ be a set of memory states where each memory state $m \in \mathcal{M}$ is an assignment of all (normal and random) variables (*Var* $\cup$ H) to values. First, an expression e of base type $\mathcal{B}$ is interpreted as a function $[\![e]\!] : m \to [\![\mathcal{B}]\!]$, where $[\![\mathcal{B}]\!]$ represents the set of values belonging to the base type $\mathcal{B}$. We omit expression semantics since it is mostly standard[4].

A random expression g is interpreted as a distribution on real values. Hence, $[\![g]\!] : \mathbf{Dist}([\![\texttt{num}]\!])$. Moreover, a command c is interpreted as a function $[\![c]\!] : \mathcal{M} \to \mathbf{Dist}(\mathcal{M})$. For brevity, we write $[\![e]\!]_m$ and $[\![c]\!]_m$ instead of $[\![e]\!](m)$ and $[\![c]\!](m)$ hereafter. Figure 5 provides the semantics of commands, where functions `unit`

[3] We use this trivial algorithm here for its simplicity. We will analyze a more sophisticated version in Section 6.

[4] The reals in LightDP only come from sampling (or, the havoc command, which mimics sampling). We assume the sample space is either finite or countable.

$$
\begin{aligned}
[\![\texttt{skip}]\!]_m &= \texttt{unit}\ m \\
[\![x := e]\!]_m &= \texttt{unit}\ (m\{[\![e]\!]_m/x\}) \\
[\![\eta := g]\!]_m &= \texttt{bind}\ [\![g]\!]\ (\lambda v.\ \texttt{unit}\ m\{v/\eta\}) \\
[\![c_1; c_2]\!]_m &= \texttt{bind}\ ([\![c_1]\!]_m)\ [\![c_2]\!] \\
[\![\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]_m &= \begin{cases} [\![c_1]\!]_m & \text{if } [\![e]\!]_m = \texttt{true} \\ [\![c_2]\!]_m & \text{if } [\![e]\!]_m = \texttt{false} \end{cases} \\
[\![\texttt{while}\ e\ \texttt{do}\ c]\!]_m &= w^*\ m \\
\text{where } w^* &= fix(\lambda f.\ \lambda m.\texttt{if}\ [\![e]\!]_m = \texttt{true} \\
&\qquad \texttt{then}\ (\texttt{bind}\ [\![c]\!]_m\ f)\ \texttt{else}\ (\texttt{unit}\ m)) \\
[\![c; \texttt{return}\ e]\!]_m &= \texttt{bind}\ ([\![c]\!]_m)\ (\lambda m'.\ \texttt{unit}\ [\![e]\!]_{m'})
\end{aligned}
$$

Figure 5. LightDP$_0$: language semantics.

and `bind` are defined in Section 2.1. This semantics corresponds directly to a semantics given by Kozen (1981), which interprets programs as continuous linear operators on measures.

Finally, we assume all programs have the form $(c; \texttt{return}\ e)$ where c does not contain return statements. A LightDP$_0$ program is interpreted as a function $m \to \mathbf{Dist}[\![\mathcal{B}]\!]$, defined in Figure 5, where $\mathcal{B}$ is the type of expression returned (e).

3.3 Typing Rules and Target Language

We assume a typing environment Γ that tracks the type of each variable (including random variable). For now, we assume a type annotation is provided for each variable (i.e., $\texttt{dom}(\Gamma) = \mathit{Var} \cup \mathrm{H}$), but we will remove this restriction in Section 5. The typing rules are formalized in Figure 6. Since all typing rules share a global invariant Ψ (e.g., the precondition in Figure 1), typing rules do not propagate Ψ for brevity. We also write $\Gamma(x) = \mathbb{d}$ for $\exists \mathcal{B}.\ \Gamma(x) = \mathcal{B}_{\mathbb{d}}$ when the context is clear.

Expressions For expressions, each rule has the form of $\Gamma \vdash e : \tau$, meaning that the expression e has type τ under the environment Γ. Rule (T-OPLUS) precisely tracks the distance of linear operations (e.g., $+$ and $-$), while rule (T-OTIMES) makes a conservative assumption that other numerical operations take identical parameters. It is completely possible to refine rule (T-OTIMES)) (e.g., by following the sensitivity analysis proposed by Reed and Pierce (2010); Gaboardi et al. (2013)) to improve precision, however, we leave that as future work since it is largely orthogonal.

Rule (T-VARSTAR) applies when variable x has a star type. This rule unpacks the corresponding pair with Sigma type and makes $\widehat{x}$ explicit in the type system.

The most interesting and novel rule is (T-ODOT). It type-checks a comparison of two real expressions by generating a constraint:

$$\Psi \Rightarrow (e_1 \odot e_2 \Leftrightarrow (e_1 + \mathbb{d}_1) \odot (e_2 + \mathbb{d}_2))$$

Intuitively, this constraint requires that in two related executions, the Boolean value of $e_1 \odot e_2$ must be identical since the distances of e_1 and e_2 are specified by $\mathbb{d}_1$ and $\mathbb{d}_2$ respectively. For example, consider the branch condition $q[i] + \eta_2 \geq \tilde{T}$ in Figure 1. Rule (T-ODOT) first checks types for subexpressions:

$$\Gamma \vdash q[i] + \eta_2 : \texttt{num}_{\widehat{q[i]} + (q[i] + \eta_2 \geq \tilde{T} ? 2 : 0)} \text{ and } \Gamma \vdash \tilde{T} : \texttt{num}_1$$

Then the following constraint is generated (free variables in the generated constraint are universally quantified):

$$\forall q_i, \widehat{q_i}, \eta_2, \tilde{T} \in \mathbb{R}.\ (-1 \leq \widehat{q_i} \leq 1) \Rightarrow$$
$$q_i + \eta_2 \geq \tilde{T} \Leftrightarrow q_i + \eta_2 + \widehat{q_i} + (q_i + \eta_2 \geq \tilde{T} ? 2 : 0) \geq \tilde{T} + 1$$

Typing rules for expressions.

$$\frac{}{\Gamma \vdash r : \texttt{num}_0}\ \text{(T-NUM)} \qquad \frac{}{\Gamma \vdash b : \texttt{bool}}\ \text{(T-BOOLEAN)}$$

$$\frac{}{\Gamma, x : \mathcal{B}_{\mathbb{d}} \vdash x : \mathcal{B}_{\mathbb{d}}}\ \text{(T-VAR)} \qquad \frac{}{\Gamma, x : \mathcal{B}_* \vdash x : \mathcal{B}_{\widehat{x}}}\ \text{(T-VARSTAR)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num}_{\mathbb{d}_1} \quad \Gamma \vdash e_2 : \texttt{num}_{\mathbb{d}_2}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{num}_{\mathbb{d}_1 \oplus \mathbb{d}_2}}\ \text{(T-OPLUS)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num}_0 \quad \Gamma \vdash e_2 : \texttt{num}_0}{\Gamma \vdash e_1 \otimes e_2 : \texttt{num}_0}\ \text{(T-OTIMES)}$$

$$\frac{\begin{array}{ll} \Gamma \vdash e_1 : \texttt{num}_{\mathbb{d}_1} & \Psi \Rightarrow (e_1 \odot e_2 \\ \Gamma \vdash e_2 : \texttt{num}_{\mathbb{d}_2} & \Leftrightarrow (e_1 + \mathbb{d}_1) \odot (e_2 + \mathbb{d}_2)) \end{array}}{\Gamma \vdash e_1 \odot e_2 : \texttt{bool}}\ \text{(T-ODOT)}$$

$$\frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \neg e : \texttt{bool}}\ \text{(T-NEG)} \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \texttt{list}\ \tau}{\Gamma \vdash e_1 :: e_2 : \texttt{list}\ \tau}\ \text{(T-CONS)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{list}\ \tau \quad \Gamma \vdash e_2 : \texttt{num}_0}{\Gamma \vdash e_1[e_2] : \tau}\ \text{(T-INDEX)}$$

Typing rules for commands

$$\frac{}{\Gamma \vdash \texttt{skip} \rightharpoonup \texttt{skip}}\ \text{(T-SKIP)}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash x : \mathcal{B}_{\mathbb{d}} \quad \tau = \mathcal{B}_{\mathbb{d}}}{\Gamma \vdash x := e \rightharpoonup x := e}\ \text{(T-ASGN)}$$

$$\frac{\Gamma \vdash x : \mathcal{B}_{\widehat{x}} \qquad \Gamma \vdash e : \mathcal{B}_{\mathbb{d}}}{\Gamma \vdash x := e \rightharpoonup x := e; \widehat{x} := \mathbb{d};}\ \text{(T-ASGNSTAR)}$$

$$\frac{\Gamma \vdash c_1 \rightharpoonup c_1' \quad \Gamma \vdash c_2 \rightharpoonup c_2'}{\Gamma \vdash c_1; c_2 \rightharpoonup c_1'; c_2'}\ \text{(T-SEQ)}$$

$$\frac{\Gamma \vdash e : \texttt{num}_0 \quad \mathit{or} \quad \Gamma \vdash e : \texttt{bool}_0}{\Gamma \vdash \texttt{return}\ e \rightharpoonup \texttt{return}\ e}\ \text{(T-RETURN)}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash c_i \rightharpoonup c_i' \text{ where } i \in \{1, 2\}}{\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \rightharpoonup \texttt{if}\ e\ \texttt{then}\ c_1'\ \texttt{else}\ c_2'}\ \text{(T-IF)}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \qquad \Gamma \vdash c \rightharpoonup c'}{\Gamma \vdash \texttt{while}\ e\ \texttt{do}\ c \rightharpoonup \texttt{while}\ e\ \texttt{do}\ c'}\ \text{(T-WHILE)}$$

Typing rules for random assignments

$$\frac{\Gamma(\eta) = \texttt{num}_{\mathbb{d}}}{\Gamma \vdash \eta := \texttt{Lap}\ r \rightharpoonup \texttt{havoc}\ \eta; \mathbf{v}_\epsilon = \mathbf{v}_\epsilon + |\mathbb{d}|/r;}\ \text{(T-LAPLACE)}$$

Figure 6. Typing rules. Ψ is an invariant that holds throughout program execution.

$$
\begin{array}{lll}
\text{Vars} & x & \in \ \mathit{Var} \cup \mathrm{H} \cup \{\mathbf{v}_\epsilon\} \\
\text{Statements} & c & ::= \ \texttt{skip} \mid x := e \mid \texttt{havoc}\ x \mid c_1; c_2 \mid \texttt{return}\ e \\
& & \quad\ \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c
\end{array}
$$

Figure 7. Target language syntax. The omitted parts are identical to the source language defined in Figure 3.

This proof obligation captures a subtle yet important property of the Sparse Vector method: given the randomness alignment as specified by Γ, two related executions must take the same branch (hence, produce the same output). This proof obligation can easily be discharged by an external SMT solver, such as Z3 (de Moura and Bjørner 2008).

Target language The typing rules for a command have the form of $\Gamma \vdash c \rightharpoonup c'$, where c is the original program being verified, and c' is the transformed program in the target language defined in Figure 7. The target language is mostly identical to the original one, except for two significant differences: 1) the target language involves a distinguished variables $\mathbf{v}_\epsilon$ to explicitly track the privacy cost in the original program; 2) the target language removes probabilistic expressions, and introduces a new nondeterministic command ($\texttt{havoc}\ x$), which sets variable x to an arbitrary value upon execution. Hence, the target language is nonprobabilistic.

Due to nondeterminism, the denotational semantics interprets a command c in the target language as a function $[\![c]\!] : \mathcal{M} \to \mathcal{P}(\mathcal{M})$. For example, the semantics of the `havoc` command is defined as follows:

$$[\![\texttt{havoc}\ x]\!]_m = \cup_{r \in \mathbb{R}} \{m\{r/x\}\}$$

Other commands have a standard semantics, hence their semantics are included in the full version of this paper (Zhang and Kifer 2016). Note that for simplicity, we abuse the notation $[\![c]\!]$ to denote the semantics of both the source language and target language. However, its meaning is unambiguous in the context of a memory m: when $\mathbf{v}_\epsilon \notin \texttt{dom}(m)$, $[\![c]\!]_m$ denotes a distribution; otherwise, $[\![c]\!]_m$ denotes a set.

Commands Informally, if $\Gamma \vdash c \rightharpoonup c'$, and the distinguished variable $\mathbf{v}_\epsilon$ in c' is bounded by some constant ϵ' in all possible executions, then program c is ϵ'-differentially private. Here, we discuss important typing rules to enforce this property. We will formalize this soundness property and sketch a proof in Section 4.

For an assignment $x := e$, rule (T-ASGN) *synthesizes* the types of x and e, and *checks* that their types are equivalent (i.e., both the base type and distance are equivalent). However, rule (T-ASGNSTAR) instruments the original program so that the typing invariant (i.e., the distance of x is exactly $\widehat{x}$) is maintained after the assignment. Consider line 4 in Figure 4. Rule (T-ASGNSTAR) first checks subexpressions: $\Gamma \vdash \texttt{sum} + q[i] : \widehat{\texttt{sum}} + \widehat{q}[i]$. Hence, the transformed program is $(\texttt{sum} := \texttt{sum} + q[i]; \widehat{\texttt{sum}} := \widehat{\texttt{sum}} + \widehat{q}[i])$, which correctly maintains the typing invariant after the assignment.

(T-RETURN) checks that the returned value is indistinguishable in two related executions. Both (T-IF) and (T-WHILE) check that two related executions must follow the same control flow.

Laplace mechanism Intuitively, (T-LAPLACE) assigns a polymorphic type to the random source $\texttt{Lap}\ r$. In other words, for any distance $\mathbb{d}$ of random variable η, we can instantiate the type of $\texttt{Lap}\ r$ to be $\texttt{num}_{\mathbb{d}}$, though with a privacy cost of $|\mathbb{d}|/r$. Moreover, the transformed program sets η to a nondeterministic value ($\texttt{havoc}\ \eta$), since any real value can be sampled from the Laplace distribution.

Consider line 1 in Figure 1. (T-LAPLACE) transforms this line to $(\texttt{havoc}\ \eta_1; \mathbf{v}_\epsilon := \mathbf{v}_\epsilon + \epsilon/2)$ since $\Gamma(\eta_1) = \texttt{num}_1$. Informally, the transformation says that by paying a privacy cost of $\epsilon/2$, $\texttt{Lap}\ (2/\epsilon)$ ensures that η_1 has a distance of one in two related executions.

Moreover, consider line 6 in Figure 4. (T-LAPLACE) transforms this line to $(\texttt{havoc}\ \eta; \mathbf{v}_\epsilon := \mathbf{v}_\epsilon + |\widehat{\texttt{sum}}|\epsilon/b)$ since $\Gamma(\eta) = -\widehat{\texttt{sum}}$. Informally, the transformation says that by paying a privacy cost, η has a distance of $-\widehat{\texttt{sum}}$ in two related executions. Hence, we can cancel out the distance of `sum` in line 7.

Dependent types and imperative programming Mutable states in imperative programming brings subtleties that are not foreseen in the standard theory of dependent types (Martin-Löf 1984). Consider a variable x with type $\texttt{num}_y$ where y is initialized to zero. The type of x establishes an invariant on its values v_1, v_2 under two executions: $v^{(1)} = v^{(2)}$. However, if we update the value of y to 1, the invariant changes to $v^{(1)} + 1 = v^{(2)}$, but the values of x in two executions remains unchanged. Hence, the type invariant is broken.

To address this issue, we assume the following assumptions are checked before type checking. First, for each normal variable $x \in \mathit{Var}$ such that $\Gamma(x) = \mathcal{B}_{\mathbb{d}}$, all free variables in $\mathbb{d}$ are immutable. For example, each normal variables in Figure 1 has a constant distance in its type. Note that by language syntax, this restriction does not apply to variables with a star type. Second, a random variable $\eta \in \mathrm{H}$ may depend on mutable variables. However, we assume that it has only *one use* other than the definition, and the definition of η is adjacent to its use. Hence, each variable that η depends on appears immutable between η's definition and use [5].

4. Soundness

The type system in Section 3.3 enforces a fundamental property: if $\Gamma \vdash c \rightharpoonup c'$ and $\mathbf{v}_\epsilon$ in c' is bounded by some constant ϵ, then the original program being verified is ϵ-differentially private.

To formalize and prove this soundness property, we first notice that a typing environment Γ defines a relation on two memories, since Γ specifies the exact distance of each variable:

Definition 2 (Γ-Relation)**.** *Two memories m_1 and m_2 are related by a typing environment Γ, written $m_1\ \Gamma\ m_2$, iff*

$$\forall x \in \text{Var}.\ m_1(x) + [\![\mathbb{d}_x]\!]_{m_1} = m_2(x),\ \textit{where}\ \Gamma \vdash x : \mathcal{B}_{\mathbb{d}_x}$$

Note that since $\Gamma(x)$ might be a dependent type, the definition needs to evaluate the distance of x ($\mathbb{d}_x$) under m_1.

By the definition above, Γ is a function since for any memory m, the distance for each variable in the related memory of m is a constant. Hence, we also write $\Gamma(m)$ to represent the unique m' such that $m\ \Gamma\ m'$. Moreover, given a set of distinct memories $S \subseteq \mathcal{M}$, we define $\Gamma\ S \triangleq \{\Gamma(m) \mid m \in S\}$. Note that by definition, $\Gamma\ S$ is also a set of distinct memories (hence, *not* a multiset). Furthermore, we assume that Γ is an injective function. We make this assumption explicit by the following definition.

Definition 3 (Well-Formed Γ-relation)**.** *A typing environment Γ is well-formed, written $\vdash \Gamma$, iff Γ is an injective function.*

Checking well-formedness of the Γ-relation is straightforward. Intuitively, Γ is well-formed when there is no "circular" dependency, while more careful analysis is needed for circular dependencies. Consider the Sparse Vector method in Figure 1 and any m_1 and m_2 such that $\Gamma\ m_1 = \Gamma\ m_2 = m$ for some m. For a variable y with a constant distance $\mathtt{v}$ (e.g., $\tilde{T}, \eta_1, \widehat{q}[i]$), we have $m_1(y) = m(y) - \mathtt{v} = m_2(y)$. So m_1 and m_2 must agree on those variables. Then for any variable z that depends on variables that m_1 and m_2 already agree on, the distance of z must be

[5] The one-use assumption may appear restrictive at first glance, but when η's type has no dependency on mutable variables, we can always store η to a normal variable to circumvent this restriction. When η's type depends on a mutable variable and multiple uses of η are needed, we can store η's value to a normal variable with star type, whose distance counterpart is manipulated and reasoned about by the type system.

identical in m_1 and m_2; hence, $m_1(z) = m_2(z)$. For the circular dependency on variable η_2 (whose distance depends η_2), consider $\mathtt{t} = [\![\tilde{T} - q[i]]\!]_{m_1} = [\![\tilde{T} - q[i]]\!]_{m_2}$. The mapping for η_2 is $v \mapsto v + 2$ when $v > \mathtt{t}$ and $v \mapsto v$ otherwise. Since this mapping is strictly monotonic, it is injective.

For differential privacy, we are interested in the relationship between two *memory distributions*. Given a typing environment Γ and constant ϵ, we define the (Γ, ϵ) distance, written $\Delta\Gamma_\epsilon$, of two memory distributions:

Definition 4 (Γ_ϵ-distance)**.** *The Γ_ϵ-distance of two distributions $\mu_1, \mu_2 \in \mathbf{Dist}(\mathcal{M})$, written $\Delta\Gamma_\epsilon(\mu_1, \mu_2)$, is defined as:*

$$\Delta\Gamma_\epsilon(\mu_1, \mu_2) \triangleq \max_{S \subseteq \mathcal{M}} (\mu_1(S) - \exp(\epsilon)\mu_2(\Gamma\ S))$$

Note that when $S = \emptyset$, the distance is 0 by definition. So $\Delta\Gamma_\epsilon(\mu_1, \mu_2) \geq 0$ for any ϵ, μ_1, μ_2.

The soundness theorem connects the "privacy cost" of the probabilistic program to the distinguished variable $\mathbf{v}_\epsilon$ in the transformed nonprobabilistic program. In order to formalize the connection, we first extend memory in the source language to include $\mathbf{v}_\epsilon$:

Definition 5. *For any memory m and constant ϵ, there is an extension of m, written $m \uplus (\epsilon)$, so that*

$$\forall x \in \mathtt{dom}(m).\ m \uplus (\epsilon)(x) = m(x)$$
$$\wedge\ m \uplus (\epsilon)(\mathbf{v}_\epsilon) = \epsilon$$

Next, we introduce useful lemmas and theorems. First, we show that the type-directed transformation $\Gamma \vdash c \rightharpoonup c'$ is *faithful*. In other words, for any initial memory m and program c, memory m' is a possible final memory iff for initial extended memory $m \uplus (0)$ and c', one final memory is an extension of m'.

Lemma 1 (Faithfulness)**.**

$$\forall m, m', c, c', \Gamma.\ \Gamma \vdash c \rightharpoonup c' \Rightarrow$$
$$[\![c]\!]_m(m') \neq 0 \Leftrightarrow \exists \epsilon.\ m' \uplus (\epsilon) \in [\![c']\!]_{m \uplus (0)}$$

Proof. By structural induction on c. □

For a pair of initial and final memories m_0 and m' when executing the original program, we identify a set of possible $\mathbf{v}_\epsilon$ values, so that in the corresponding executions of c', the initial and final memories are extensions of m and m' respectively:

Definition 6. *Given a target program c', an initial memory m_0 and a final memory m', the consistent costs of executing c' w.r.t. m_0 and m', written $c' \restriction^{m'}_{m_0}$, is defined as follows*

$$c' \restriction^{m'}_{m_0} \triangleq \{\epsilon' \mid m \uplus (\epsilon') \in [\![c']\!]_{m_0 \uplus (0)} \wedge m = m'\}$$

where $m = m'$ iff $\forall x \in \mathtt{dom}(m).\ m'(x) = m(x)$

Since $(c' \restriction^m_{m_1})$ by definition is a set of values of $\mathbf{v}_\epsilon$, we write $\max(c' \restriction^m_{m_1})$ for the maximum cost. The next lemma enables precise reasoning of privacy cost w.r.t. a pair of initial and final memories when Γ is injective:

Lemma 2 (Point-Wise soundness)**.**

$$\forall c, c', m_1, m_2, m, \Gamma.\ \vdash \Gamma \wedge \Gamma \vdash c \rightharpoonup c' \wedge m_1\ \Gamma\ m_2,\ \textit{we have}$$
$$[\![c]\!]_{m_1}(m) \leq \exp(\max(c' \restriction^m_{m_1}))[\![c]\!]_{m_2}(\Gamma(m))$$

The full proof of Lemma 2 is available in the full version of this paper (Zhang and Kifer 2016). We comment that this point-wise result enables precise reasoning of privacy cost where the composition theorem falls short. Consider the transformed Sparse Vector method in Figure 2. This point-wise result allows various cost bounds to be provided for various memories: $\mathbf{v}_\epsilon$ increments by $2\epsilon/N$ when the branch condition is true, but it remains the same otherwise. On the other hand, methods based on the composition theorem (e.g., (Reed and Pierce 2010; Gaboardi et al. 2013; Barthe et al. 2012, 2014)) have to (conservatively) provide an unique cost bound for all possible executions, rendering a cost of $2\epsilon/N$.

The point-wise soundness lemma provides a precise privacy bound per initial and final memory. However, differential privacy by definition (Definition 1) bounds the worst-case cost. To close the gap, we define the worst-case cost of the transformed program.

Definition 7. *For any program c in the target language, we say c's execution cost is* bounded *by some constants ϵ, written $c^{\preceq\epsilon}$, iff for* any $m \uplus (0)$,

$$m' \uplus (\epsilon') \in [\![c]\!]_{m \uplus (0)} \Rightarrow \epsilon' \leq \epsilon$$

Note that this safety property can be verified by an external mechanism such as Hoare logic and model checking. Off-the-shelf tools can be used to verify that $c^{\preceq\epsilon}$ holds for some ϵ. For example, we have formally proved that the transformed program in Figure 1 satisfies a postcondition $\mathbf{v}_\epsilon \leq \epsilon$ by providing one line of annotation (the grey line in Figure 1) using the Dafny tool (Leino 2010).

Theorem 1 (Soundness)**.**

$$\forall c, c', m_1, m_2, \Gamma, \epsilon.\ \vdash \Gamma \wedge \Gamma \vdash c \rightharpoonup c' \wedge m_1\ \Gamma\ m_2 \wedge c'^{\preceq\epsilon},\ \textit{we have}$$
$$\Delta\Gamma_\epsilon([\![c]\!]_{m_1}, [\![c]\!]_{m_2}) \leq 0$$

Proof. By definition, $(\max(c \restriction^m_{m_1})) \leq \epsilon$ for all m, m_1. Hence by Lemma 2, $\forall m.\ [\![c]\!]_{m_1}(m) \leq \exp(\epsilon)[\![c]\!]_{m_2}(\Gamma(m))$. Hence,

$$\max_{S \subseteq \mathcal{M}} ([\![c]\!]_{m_1}(S) - \exp(\epsilon)[\![c]\!]_{m_2}(\Gamma(S)))$$
$$= \max_{S \subseteq \mathcal{M}} \sum_{m \in S} ([\![c]\!]_{m_1}(m) - \exp(\epsilon)[\![c]\!]_{m_2}(\Gamma(m))) \leq 0$$

We note that the equality in the proof above holds due to the injective assumption ($\vdash \Gamma$), which allows us to derive the set-based privacy from the point-wise privacy (Lemma 2). □

We now connect the soundness theorem to differential privacy:

Theorem 2 (Privacy)**.**

$$\forall \Gamma, c, c', x, \epsilon.\ \vdash \Gamma \wedge \Gamma \vdash (c; \mathtt{return}\ e) \rightharpoonup (c'; \mathtt{return}\ e)\ \textit{then}$$
$$c'^{\preceq\epsilon} \Rightarrow c\ \textit{is}\ \epsilon\textit{-differentially private}$$

Proof. Proof is available in the full version of this paper (Zhang and Kifer 2016). □

5. Differential-Privacy Proof Inference

We have so far presented an explicitly typed language LightDP$_0$. However, writing down types (especially those dependent types) for variables is still a non-trivial task. Moreover, when multiple proofs exist, writing down types accompanied with the minimum privacy cost is even more challenging. We extend LightDP$_0$ to automatically infer a proof and even search for the optimal one.

5.1 Type Inference

Since each type has two orthogonal components (base type and distance), inference is needed for both. The former is mostly standard (e.g., for Hindley/Milner system (Wand 1987; Aiken and Wimmers 1993; Zhang and Myers 2014)), hence omitted in this paper.

Distance Vars $\alpha, \beta, \gamma \in$ *DVar*
Distances $\quad \mathbb{d} \quad ::= \cdots \mid \alpha$

Figure 8. LightDP: language syntax extension.

Next, we assume all base types are available, and focus on the inference of the distance counterpart. For brevity, we write $\Gamma(x) = \mathbb{d}$ instead of $\exists \mathcal{B}.\, \Gamma(x) = \mathcal{B}_{\mathbb{d}}$. We use `DefVars` to represent the set of variables whose distances are given by the programmer.

To enable type inference, we extend LightDP$_0$ with distance variables such as α, β, γ (shown in Figure 8). Initially, the typing environment associates each variable in `DefVars` with its annotated distance. It associates each other variable with a distinguished distance variable to be inferred.

Following the idea of modeling type inference as constraint solving (e.g., (Wand 1987; Aiken and Wimmers 1993; Haack and Wells 2004)), it is straightforward to interpret the typing rules in Figure 6 as a (naive) inference algorithm. To see how, consider two assignments $(x := 0; y := x)$, where $\Gamma(x) = \alpha, \Gamma(y) = \beta$. With distance variables, the typing rules now collect constraints (instead of checking their validity) during type checking. For example, two constraints are collected for those two assignments: $\alpha = 0$ and $\beta = \alpha$. Hence, inferring types is equivalent to finding a solution for those two constraints (i.e., the satisfiability problem of $\exists \alpha, \beta.\ \alpha = 0 \wedge \beta = \alpha$). It is easy to check that $\alpha = 0 \wedge \beta = 0$ is a solution. Hence, the inferred distances are $\Gamma(x) = 0, \Gamma(y) = 0$. However, this naive inference algorithm falls short in face of dependent types. Next, we first explore the main challenges in inferring dependent types, and then propose our inference algorithm.

Inferring star types Consider the example in Figure 4. If we follow the naive inference algorithm above, two constraints are generated from lines 2 and 4: $\alpha = 0$ and $\alpha = \alpha + \widehat{q}[i]$, where $\alpha = \Gamma(\texttt{sum})$. These constraints are unsatisfiable, since the value of $\widehat{q}[i]$ is an arbitrary value between -1 and 1. Nevertheless, the powerful type system of LightDP$_0$ still allows formal verification of this example by assigning `sum` to the star type, meaning that its distance is dynamically tracked.

We observe that starting from the initial typing environment, we can refine it by processing each assignment $x := e$ in the following way. We first *synthesize* the type of e from its subexpressions, in the same fashion as the original typing rules in Figure 6. Then, if $x \in$ `DefVars` (i.e., given by the programmer), there is nothing to be refined. Otherwise, we can *refine* the typing environment by updating the type of x to a more precise one:

$$\texttt{refine}(\Gamma, x, \mathbb{d}) \triangleq \begin{cases} \Gamma\{\mathbb{d}/\alpha\} & \text{if } \Gamma(x) = \alpha \in \textit{DVar} \\ \Gamma & \text{if } \Gamma(x) \notin \textit{DVar} \wedge (\Gamma(x) = \mathbb{d}) \\ \Gamma[x \mapsto *] & \text{otherwise} \end{cases}$$

Here, the auxiliary function `refine` takes an initial environment Γ, a variable x and a distance expression $\mathbb{d}$. This function replaces all occurrences of α in Γ to $\mathbb{d}$ when $\Gamma(x)$ is a variable to be inferred ($\alpha \in \textit{DVar}$). Otherwise, it statically checks whether the old and new distance expressions are equivalent. When the equivalence cannot be determined at static time, it assigns the $*$ type to x.

Our inference algorithm refines the typing environment as it proceeds. Consider Figure 4 again. At line 4, `sum`'s distance is refined to 0. Then at line 6, its distance is refined to $*$, since we cannot statically check that $0 = 0 + \widehat{q}[i]$ is valid.

Inferring dependency on program state Consider Figure 1 where only the type of η_2 is to be inferred. The naive inference algorithm will generate one constraint for the branch condition in line 6:

$$\forall q_i, \widehat{q_i}, \eta_2, \tilde{T}.\ (-1 \le \widehat{q_i} \le 1) \Rightarrow (q_i + \eta_2 \ge \tilde{T} \Leftrightarrow (q_i + \eta_2 + \widehat{q_i} + \alpha \ge \tilde{T} + 1))$$

which is unsatisfiable, since there is no single value α that can hide the difference of q_i in both directions. We need a more precise type for η_2 (as provided in Figure 1) so that the "if" and "else" branches can be aligned in different ways.

Refinement rules for expressions

$$\frac{}{\Gamma; \mathcal{P} \bowtie r : \Gamma}\ \text{(R-Num)} \qquad \frac{b \in \{\texttt{true}, \texttt{false}\}}{\Gamma; \mathcal{P} \bowtie b : \Gamma}\ \text{(R-Boolean)}$$

$$\frac{}{\Gamma; \mathcal{P} \bowtie x : \Gamma}\ \text{(R-Var)} \qquad \frac{\mathcal{P} = \emptyset}{\Gamma; \mathcal{P} \bowtie \eta : \Gamma}\ \text{(R-Rand)}$$

$$\frac{\mathcal{P} \neq \emptyset \qquad \alpha_t, \alpha_f \text{ fresh variables}}{\Gamma; \mathcal{P} \bowtie \eta : \texttt{refine}(\Gamma, \eta, \mathcal{P}?\alpha_t : \alpha_f)}\ \text{(R-Rand-Refine)}$$

$$\frac{\Gamma; \mathcal{P} \bowtie e_1 : \Gamma_1 \quad \Gamma_1; \mathcal{P} \bowtie e_2 : \Gamma_2 \quad \texttt{op} \in \oplus \cup \otimes}{\Gamma; \mathcal{P} \bowtie e_1\ \texttt{op}\ e_2 : \Gamma_2}\ \text{(R-Ops)}$$

$$\frac{\Gamma; \mathcal{P} \wedge (e_1 \odot e_2) \bowtie e_1 : \Gamma_1 \quad \Gamma_1; \mathcal{P} \wedge (e_1 \odot e_2) \bowtie e_2 : \Gamma_2}{\Gamma; \mathcal{P} \bowtie e_1 \odot e_2 : \Gamma_2}\ \text{(R-ODot)}$$

$$\frac{\Gamma; \mathcal{P} \bowtie e_1 : \Gamma_1 \quad \Gamma_1; \mathcal{P} \bowtie e_2 : \Gamma_2}{\Gamma; \mathcal{P} \bowtie e_1 :: e_2 : \Gamma_2}\ \text{(R-Cons)}$$

$$\frac{\Gamma; \mathcal{P} \bowtie e : \Gamma'}{\Gamma; \mathcal{P} \bowtie \neg e : \Gamma'}\ \text{(R-Neg)} \qquad \frac{\Gamma; \mathcal{P} \bowtie e_1 : \Gamma_1 \quad \Gamma_1; \mathcal{P} \bowtie e_2 : \Gamma_2}{\Gamma; \mathcal{P} \bowtie e_1[e_2] : \Gamma_2}\ \text{(R-Idx)}$$

Refinement rules for commands

$$\frac{}{\Gamma \bowtie \texttt{skip} : \Gamma}\ \text{(R-Skip)} \qquad \frac{}{\Gamma \bowtie \texttt{return}\ e : \Gamma}\ \text{(R-Return)}$$

$$\frac{x \notin \texttt{DefVars} \qquad \Gamma, \emptyset \bowtie e : \Gamma' \qquad \Gamma' \vdash e : \mathbb{d}}{\Gamma \bowtie x := e : \texttt{refine}(\Gamma', x, \mathbb{d})}\ \text{(R-Asgn-Ref)}$$

$$\frac{x \in \texttt{DefVars} \quad \Gamma, \emptyset \bowtie e : \Gamma'}{\Gamma \bowtie x := e : \Gamma'}\ \text{(R-Asgn)}$$

$$\frac{\Gamma \bowtie c_1 : \Gamma' \quad \Gamma' \bowtie c_2 : \Gamma''}{\Gamma \bowtie c_1; c_2 : \Gamma''}\ \text{(R-Seq)}$$

$$\frac{\Gamma; \emptyset \bowtie e : \Gamma_1 \quad \Gamma_1 \bowtie c_1 : \Gamma_2 \quad \Gamma_2 \bowtie c_2 : \Gamma_3}{\Gamma \bowtie \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \Gamma_3}\ \text{(R-If)}$$

$$\frac{\Gamma; \emptyset \bowtie e : \Gamma_1 \qquad \Gamma_1 \le \Gamma_2 \qquad \Gamma_2 \bowtie c : \Gamma_2}{\Gamma \bowtie \texttt{while}\ e\ \texttt{do}\ c : \Gamma_2}\ \text{(R-While)}$$

Refinement for random assignments

$$\frac{}{\Gamma \bowtie \eta := \texttt{Lap}\ r : \Gamma}\ \text{(R-Laplace)}$$

Figure 9. The refinement algorithm.

To infer dependent types, our inference algorithm propagates context information to subexpressions. In particular, we observe that only rule (T-ODot) generates a constraint that may benefit from dependency on program states. Hence, our inference algorithm propagates the comparison result to its subexpressions, and refine subexpressions (e.g., η_2) for the needed dependency.

Inference algorithm We now present our inference algorithm, which is still based on the typing rules in Figure 6. However, to tackle the challenges above, we run a *refinement algorithm* before type inference. The algorithm is shown in Figure 9.

For expressions, the refinement algorithm propagates context information $\mathcal{P}$ to subexpressions. Hence, each rule for expression has the form of $\Gamma, \mathcal{P} \bowtie e : \Gamma'$, where $\mathcal{P}$ is a predicate that may appear in a dependent type, Γ is the typing environment to be refined, and Γ' is the refined environment. The context information $\mathcal{P}$ is used to refine distance of a random variable η in rule (R-RAND-REFINE). Note that the refinement is not needed for a normal variable x (rule (R-VAR)). Intuitively, the reason is that the "shape" of x is either provided or has been refined when x is initialized. However, this is not true for a random variable: η can have any distance expression according to rule (T-LAPLACE).

The refinement rules for commands have the form of $\Gamma \bowtie c : \Gamma'$. As we described informally above, rule (R-ASGN-REF) refines the distance of x using the `refine` function when its distance is not given. The rule (R-WHILE) assumes that a fixed point exists. Based on the definition of the `refine` function, a fixed point can be computed as follows. We define $\leq$ as the lifted relation based on a point-wise lattice (for each variable) where: $\forall \alpha, \beta \in \mathit{DVar}.\ \alpha \leq \beta \wedge \beta \leq \alpha$ and $\alpha \leq \mathbb{d} \leq *$ if $\mathbb{d}$ is not a distance variable. We can compute a fixed point by $\Gamma_1 = \Gamma^0 \vdash c : \Gamma^1, \Gamma^1 \vdash c : \Gamma^2, \cdots$ until $\Gamma^i \vdash c : \Gamma^i$ for some i. Based on the definition of the `refine` function, it is easy to check that $\Gamma^i \leq \Gamma^{i+1}$ and the computation terminates since whenever $\Gamma^i \neq \Gamma^{i+1}$, either the number of distance variables is reduced by one, or one more variable has a star type.

Example We consider type inference for our running example in Figure 1 where all local variables are to be inferred. We first run the refinement algorithm. The first refinement happens at line 2, where the distance of $\tilde{T}$ is refined to α, the distance variable of η_1. At line 3, $\mathtt{c}_1$, $\mathtt{c}_2$ and `i` are refined to distance 0. In the loop body, η_2 is refined to $q[i] + \eta_2 \geq \tilde{T}?\beta : \gamma$ at line 6, using rule (R-RAND-REFINE). At line 8, $\texttt{refine}(\Gamma, \mathtt{c1}, 0)$ returns Γ since $0 = 0$ is always true. Similar for the "else" branch and line 12. Hence, the environment after line 12 is already a fixed point for the loop body. Hence, the typing environment after refinement is: $\Gamma(\mathtt{c1}) = \Gamma(\mathtt{c2}) = \Gamma(i) = 0$, $\Gamma(\tilde{T}) = \Gamma(\eta_1) = \alpha$ and $\Gamma(\eta_2) = (q[i] + \eta_2 \geq \tilde{T})?\beta : \gamma$.

Type checking with distance variables With type variables in the refined environment Γ, the type system collects constraints during type checking, and tries to solve the collected constraints where the type variables are existentially qualified. For example, with type refinement, type checking the partial sum example in Figure 4 yields a unique solution, which is identical to the type annotation in the figure. In general, collected constraint may have multiple solutions. For example, type checking the Sparse Vector method generates only one (nontrivial) constraint from the rule (T-ODOT):

$$\big(\forall q_i, \widehat{q_i}, \eta_2, \tilde{T} \in \mathbb{R}.\ (-1 \leq \widehat{q_i} \leq 1) \Rightarrow$$
$$q_i + \eta_2 \geq \tilde{T} \Leftrightarrow q_i + \widehat{q_i} + \eta_2 + (q_i + \eta_2 \geq \tilde{T}?\beta : \gamma) \geq \tilde{T} + \alpha\big)$$

It is easy to check that the type annotation in Figure 1 (i.e., $\alpha = 1, \beta = 2, \gamma = 0$) is a solution of the constraint. But in fact, other solutions exist. For example, $\alpha = 0, \beta = 2, \gamma = -2$ and $\alpha = 2, \beta = 3, \gamma = 0$ are both valid solutions. The type system can either pick a solution, or defer the inference by transforming the original program to a target program where type variables are treated as unknown program inputs (as shown in Figure 10).

5.2 Minimizing Privacy Cost

With type variables captured explicitly in the transformed program, we can verify that the postcondition $\mathbf{v}_\epsilon = \frac{\alpha\epsilon}{2} + \frac{\beta\epsilon}{2} + \mathtt{c2} \times \frac{\gamma\epsilon}{2N}$ holds by providing the loop invariant shown in grey. Hence, combined with the remaining unsolved constraints on those type variables, finding the optimal proof is equivalent to the following MaxSMT problem, where M is a large number since `c2` is not bounded:

$$\min\Big(\frac{\alpha}{2} + \frac{\beta}{2} + M \times \gamma\Big) \text{ such that}$$
$$\big(\forall q_i, \widehat{q_i}, \eta_2, \tilde{T} \in \mathbb{R}.\ (-1 \leq \widehat{q_i} \leq 1) \Rightarrow$$
$$q_i + \eta_2 \geq \tilde{T} \Leftrightarrow q_i + \widehat{q_i} + \eta_2 + (q_i + \eta_2 \geq \tilde{T}?\beta : \gamma) \geq \tilde{T} + \alpha\big)$$

Using a MaxSMT solver μZ (Bjørner and Phan 2014; Bjørner et al. 2015), we successfully find the optimal solution for the type variables: $\alpha = 1, \beta = 2, \gamma = 0$. This is exactly the randomness alignment used in its formal proof (Dwork and Roth 2014).

We note that the translation to the MaxSMT problem at this stage still requires programmer efforts (e.g., identifying the cost bound involving type variables and converting the cost bound to an equivalent formula suitable for a MaxSMT solver). However, this example clearly demonstrates the potential benefits of explicitly calculating the privacy cost in the target language.

```
function MSPARSEVEC (T, N : num; q : list num; q̂ : list num;
α, β, γ : num)
        returns out : list num
        precondition ∀i. −1 ≤ (q̂[i]) ≤ 1

1   v_ε := 0;
2   havoc η1; v_ε := v_ε + (αε/2);
3   T̃ := T + η1;
4   c1 := 0; c2 := 0; i := 0;
5   while (c1 < N)
6     Invariant : c1 ≤ N ∧ v_ε = αε/2 + c1 × βε/2N + c2 × γε/2N
7     havoc η2; v_ε := v_ε + (q[i] + η2 ≥ T̃?β : γ) × ε/4c;
8     if (q[i] + η2 ≥ T̃) then
9       out:= true::out;
10      c1 := c1+1;
11    else
12      out:= false::out;
13      c2 := c2+1;
14    i := i+1;
```

Figure 10. The target program with unknown type variables. The instrumented statements are underlined.

5.3 Proof Automation

In general, a LightDP-based proof consists four steps involving manual efforts: 1) writing down the program specification (i.e., the function signature that specifies private and non-private parameters and return values), 2) writing down the type annotations for local variables, 3) verifying that the privacy cost in the transformed program is bounded by either a known budget, or (MaxSMT only) a formula involving unsolved type variables, and 4) (MaxSMT only) solving the MaxSMT problem of "min(upper bound formula) such that constraints from step 2 are satisfiable".

As most verification tools, LightDP requires a programmer to write down specification (step 1). For step 2, we find that the inference algorithm in Section 5.1 is powerful enough to automatically infer the types for the nontrivial algorithms considered in this paper[6]. For step 3 and step 4, LightDP relies on the automation in existing verification tools. We note that though LightDP currently adds no automation in step 3 and 4, separating relational reasoning from counting privacy cost and automating task 2 greatly simplifies those steps for all examples that we have seen so far. We leave systematic research in automating the entire proof as future work.

[6] The only exception is the algorithm in Section 6.3, since the algorithm uses a uniformly distributed random source which is currently absent in the inference algorithm.

function NUMSPARSEVECTOR (T, N, ϵ : $\texttt{num}_0$; q : $\texttt{list num}_*$)
returns (out : $\texttt{list num}_0$)
precondition $\forall i.\ -1 \le (\widehat{q}[i]) \le 1$

$\texttt{c1}, \texttt{c2}, \texttt{i} : \texttt{num}_0; \tilde{T}, \eta_1 : \texttt{num}_1; \eta_2 : \texttt{num}_{q[i]+\eta_2 \ge \tilde{T} ? 2:0}; \eta_3 : \texttt{num}_{-\widehat{q}[i]}$

```
1  η1 := Lap (3/ε);
2  T̃ := T + η1;
3  c1 := 0; c2 := 0; i := 0;
4  while (c1 < N)
5    η2:= Lap (6N/ε);
6    if (q[i] + η2 ≥ T̃) then
7      η3:= Lap (3N/ε);
8      out:= (q[i]+η3)::out;
9      c1 := c1 + 1;
10   else
11     out:= 0::out;
12     c2 := c2 + 1;
13   i := i+1;
```

The transformed program, where underlined commands are added by the type system. Only one annotation (loop invariant) is needed from the programmer to verify the postcondition $\mathbf{v}_\epsilon \le \epsilon$: $\texttt{c1} \le N \wedge \mathbf{v}_\epsilon = \frac{\epsilon}{3} + \texttt{c1} \times \frac{2\epsilon}{3N}$.

```
1  vε := 0;
2  havoc η1; vε := vε + ε/3;
3  T̃ := T + η1;
4  c1 := 0; c2 := 0; i := 0;
5  while (c1 < N)
6    havoc η2; vε := vε + (q[i] + η2 ≥ T̃ ? 2 : 0) × ε/6N;
7    if (q[i] + η2 ≥ T̃) then
8      havoc η3; vε := vε + |q̂[i]| × ε/3N;
9      out:= (q[i]+η3)::out;
10     c1 := c1 + 1;
11   else
12     out:= 0::out;
13     c2 := c2 + 1;
14   i := i+1
```

Figure 11. The Numerical Sparse Vector method.

6. Case Studies

6.1 Sparse Vector with Numerical Answers

We first study a numerical variant of the Sparse Vector method. The previous version (Figure 1), produces only two types of outputs for each query: `true`, meaning the query answer is probably above the threshold; and `false`, meaning that it is probably below. The numerical variant, shown in Figure 11, replaces the output `true` with a noisy query answer. It does this by drawing fresh Laplace noise and adding it to the query (Line 8).

Verification using LightDP LightDP can easily verify this numerical variant from scratch, in a very similar way as verifying the Sparse Vector method. However, here we focus on another interesting scenario of using LightDP: the programmer (or algorithm designer) has already verified the Sparse Vector method using LightDP, and she is now exploring its variations. This is a common scenario for algorithm designers. We show that since LightDP automatically fills in most proof details, exploring variations of an algorithm requires little effort.

In particular, we assume the programmer has already obtained the (optimal) types for all local variables except η_3, and the loop invariant shown in Figure 2 from the verification of the Sparse Vector method. Hence, the type inference engine only needs to infer a type for η_3, which is trivially solved to be $\texttt{num}_{-\widehat{q}[i]}$. Moreover, LightDP transforms the original program to the one on the bottom of Figure 11. To finish the proof, according to Theorem 2, it is sufficient to verify the postcondition that $\mathbf{v}_\epsilon \le \epsilon$. In fact, only one annotation (shown in Figure 11) that is very close to the one in Figure 2 is needed to finish the proof. Hence, we just proved the numerical Sparse Vector variant for (almost) free using LightDP.

Incorrect variants The numerical variant is also historically interesting since it fixes a bug in a very influential set of lecture notes (Roth 2011); these lecture notes inadverantly re-used the same noise used for the "if" test (Line 7) instead of drawing new noise when outputting the noisy query answer. In other words, Lines 5-8 in Figure 1 are replaced with:

```
η2:= Lap (2N/ε);
q̃ = q[i] + η2
if (q̃ ≥ T̃) then
  out:= (q̃)::out;
```

For this incorrect variant, the refinement algorithm refines the type of $\tilde{q}$ to be $\widehat{q}[i] + \alpha$ when $\tilde{q}$ is defined, where $\Gamma(\eta_2) = \alpha$. Moreover, during type checking, $((\tilde{q}) :: \texttt{out})$ generates a constraint $(\widehat{q}[i] + \alpha = 0)$ by rule (T-CONS). Hence, it must be true that $\Gamma(\tilde{q}) = \texttt{num}_0$ and $\Gamma(\eta_2) = \texttt{num}_{-\widehat{q}[i]}$ after type inference. Moreover, after type checking, $\eta_2 := \texttt{Lap}\ (2\texttt{N}/\epsilon)$ is transformed to

$$(\texttt{havoc}\ \eta_2; \mathbf{v}_\epsilon := \mathbf{v}_\epsilon + |\widehat{\texttt{q}}[\texttt{i}]|(\epsilon/2\texttt{N}))$$

However, we cannot prove that the incorrect variant is ϵ'-private for any ϵ'. The reason is that $\mathbf{v}_\epsilon$ in the transformed program is clearly not bounded by any constant ϵ': $\mathbf{v}_\epsilon$ increments by $\epsilon/2N$ in the worst case in each loop iteration, but the number of iterations is unbounded (when most iterations take the "else" branch).

The failure of a formal proof of the incorrect variant also sheds lights on how to fix it. For example, if we bound the number of iterations to be N, then the incorrect variant is fixed (though with a different privacy cost).

6.2 Smart Summation

We next study a smart summation algorithm verified previously (with heavy annotations) in (Barthe et al. 2012, 2014). The pseudo code, shown in Figure 12, is adapted from (Barthe et al. 2014). The goal of this smart sum algorithm is to take a finite sequence of bits $q[0], q[1], \ldots, q[T]$ and output a noisy version of their partial sum sequence: $q[0],\ q[0] + q[1],\ \ldots,\ \sum_{i=0}^{T} q[i]$. One naive approach is to add Laplace noise to each partial sum (partial implementation is shown in Figure 4). An alternative naive algorithm is to compute a noisy bit $\tilde{q}[i] = q[i] + \texttt{Lap}(1/\epsilon)$ for each i and output $\tilde{q}[0],\ \tilde{q}[0] + \tilde{q}[1],\ \ldots,\ \sum_{i=0}^{T} \tilde{q}[i]$. However, in both approaches, the noise will swamp the true counts.

A much smarter approach was proposed by Chan et al. (2011). Intuitively, their algorithm groups q into nonoverlapping blocks of size M. So block $G_1 = \{q[0], q[1], \ldots, q[M-1]\}$, $G_2 = \{q[M], q[M+1], q[M+2], \ldots, q[2M-1]\}$, etc. Then it maintains 2 levels of noisy counts: (1) the noisy bits $\tilde{q}[i] = q[i] + \texttt{Lap}(1/\epsilon)$ for each i, and (2) the noisy block sums $\tilde{G}_j = \sum_{i \in G_j} q[i] + \texttt{Lap}(1/\epsilon)$ for each block. The partial sums are computed from these noisy counts in the following way. Consider the sum of the first $\ell+1$ bits: $\sum_{i=0}^{\ell} q[i]$. We can represent $\ell+1 = xM+c$ where $x = \lfloor \frac{\ell+1}{M} \rfloor$ and $c = \ell+1 \mod M$. Hence, the noisy partial sum can be computed from the noisy sum of the first x blocks plus the remaining c noisy bits: $\tilde{G}_1 + \tilde{G}_2 + \cdots + \tilde{G}_x + \sum_{j=0}^{c-1} \tilde{q}[xB+j]$. This algorithm is shown in Figure 12. The "if" branch keeps track of block boundaries and is responsible for summing up the noisy blocks. The "else" branch is responsible for adding in the remaining loose noisy bits (once there are enough loose bits to form a new block G_j, we use its noisy sum $\tilde{G}_j$ rather than the sum of its noisy bits).

function SMARTSUM (ϵ, M, T: $\texttt{num}_0$; q: $\texttt{list num}_*$)

returns (out : $\texttt{list num}_0$)

precondition $\forall i.\ -1 \le (\widehat{q}[i]) \le 1 \wedge (\forall i.\ \widehat{q}[i] \neq 0 \Rightarrow (\forall j \neq i.\ \widehat{q}[j] = 0))$

$\texttt{next}, \texttt{n}, \texttt{i} : \texttt{num}_0; \texttt{sum} : \texttt{num}_*; \eta_1 : \texttt{num}_{-\widehat{\texttt{sum}}-\widehat{q}[i]}; \eta_2 : \texttt{num}_{-\widehat{q}[i]}$

```
1  next:=0; n:=0; i:=0; sum := 0;
2  while i ≤ T
3    if (i + 1) mod M = 0 then
4      η1 := Lap 1/ε;
5      n := n + sum + q[i] + η1;
6      next:= n;
7      sum := 0;
8      out := next::out;
9    else
10     η2 := Lap 1/ε;
11     next:= next + q[i] + η2;
12     sum := sum + q[i];
13     out := next::out;
14   i := i+1;
```

The transformed program, where underlined commands are added by the type system. Only one annotation (loop invariant) is needed from the programmer to verify the postcondition $\mathbf{v}_\epsilon \le 2\epsilon$: $(\mathbf{v}_\epsilon + \widehat{\texttt{sum}} > 0 \Rightarrow \forall j \ge i.\ \widehat{q}[j] = 0) \wedge (\widehat{\texttt{sum}} > 0 \Rightarrow \mathbf{v}_\epsilon \le \epsilon) \wedge (\widehat{\texttt{sum}} \le 1.0) \wedge (\mathbf{v}_\epsilon \le 2\epsilon)$

```
1  v_ε = 0;
2  next:=0, n:=0, i:= 0, sum:=0; sum^ := 0;
3  while i ≤ T
4     if (i + 1) mod M = 0 then
5        havoc η1; v_ε := v_ε + |sum^ + q^[i]|ε;
6        n := n + sum + q[i] + η1;
7        next:= n;
8        sum := 0;
9        sum^ := 0;
10       out := next::out;
11    else
12       havoc η2; v_ε := v_ε + |q^[i]|ε;
13       next:= next + q[i] + η2;
14       sum := sum + q[i];
15       sum^ := sum^ + |q^[i]|;
16       out := next::out;
17    i := i+1;
```

Figure 12. The SmartSum algorithm.

Assume for two adjacent databases, at most one query answer differs, and for that query, its distance is at most one (this adjacency assumption is provided as the precondition in function signature). Hence, for queries that generate the same answer on adjacent databases, no privacy cost is paid. However, privacy cost is paid twice to hide the query answers that differ: when the noisy sum for the block containing that query is computed, and when the noisy version of that query is used. Hence informally, the SmartSum algorithm satisfies 2ϵ-privacy where ϵ is a function parameter.

Verification using LightDP LightDP successfully infers the type annotations shown in the box under function signature in Figure 12. Since all type variables are only involved in equality constraints, only one solution exists. The transformed program is shown at the bottom of Figure 12.

By Theorem 2, to prove SmartSum is 2ϵ-private, it is sufficient to verify that the postcondition $\mathbf{v}_\epsilon \le 2\epsilon$ holds for the transformed program. We notice that this program maintains the loop invariant shown in Figure 12. One observation is that once the privacy cost or the distance of variable `sum` gets positive, the query that generates different answers must have been handled already. Hence, rest queries must have identical answers on adjacent databases (($\mathbf{v}_\epsilon +$ $\widehat{\texttt{sum}} > 0 \Rightarrow \forall j \ge i.\ \widehat{q}[j] = 0)$). Using the loop invariant, we formally verified the desired postcondition $\mathbf{v}_\epsilon \le 2\epsilon$ using Dafny.

function PRIVBERNOULLI (t : $\texttt{num}_*$)

returns b : bool

precondition $0 \le t \le 1 \wedge 0 \le t + \widehat{t} \le 1$

```
1  η := Uniform[0,1];
2  if (η ≤ t) then
3    b := true;
4  else
5    b := false;
```

The transformed program where underlined commands are added by the type system:

```
1  v_ε :=0;
2  havoc η; v_ε = v_ε − log(1 + (η ≤ t)?(t^ ≥ 0?0 : (t^/t))
                                  : (t^ ≤ 0?0 : (t^/t)));
3  if (η ≤ t) then
4    b := true;
5  else
6    b := false;
```

Figure 13. The PrivBernoulli algorithm.

6.3 Categorical Outputs

Until now, we have used the Laplace mechanism, which generates numerical outputs, as the primary randomization tool for ensuring differential privacy. It might seem that categorical attributes would require completely different techniques, but indeed, they can be cleanly incorporated into LightDP with a new typing rule. We briefly show how this can be done by considering a simple mechanism that takes a private-data-dependent probability t and outputs `true` with probability t and `false` with probability $1 - t$. The algorithm shown in Figure 13.

The standard trick of generating an output `true` with probability t can be done by generating an uniform [0,1] random variable x and returning `true` if $x \le t$, and `false` otherwise. This trick converts numerical randomness into categorical randomness with a notion of distance that can be aligned between executions under related databases. Generalizations to a larger output domain are routine and, in this way, can allow some instantiations of the exponential mechanism (McSherry and Talwar 2007).

To calculate the privacy cost of aligning the binary output, we need to add a single typing rule to capture the property of uniform [0,1] distribution:

$$\frac{\Gamma(\eta) = \texttt{num}_{\eta \cdot \mathbb{d}} \qquad -1 < \mathbb{d} \le 0}{\Gamma \vdash \eta := \texttt{Uniform}_{[0,1]} \rightharpoonup \texttt{havoc}_{[0,1]}\eta; \mathbf{v}_\epsilon = \mathbf{v}_\epsilon - \log(\mathbb{d} + 1)}$$

This rule requires that the random sample is aligned by a distance of $\eta \cdot \mathbb{d}$ for some $\mathbb{d}$ (i.e., we map η to $(\mathbb{d} + 1)\eta$ in the randomness alignment). Easy to check this mapping is injective. By property of uniform distribution, the privacy cost of any such assignment is $-\log(\mathbb{d} + 1)$ where $-1 \le \mathbb{d} \le 0$.

To integrate this typing rule and uniform distribution into LightDP, we need to establish that: 1) the faithfulness of the transformation, and 2) the uniform distribution satisfies Lemma 2. The former is easy to check, and we establish the latter in the full version of this paper (Zhang and Kifer 2016).

With this new typing rule for uniform distribution, we can precisely compute the privacy cost of the algorithm in Figure 13 by providing the following type for η: $\Gamma(\eta) = \eta \cdot \mathbb{d}$ where

$$\mathbb{d} = (\eta \le t)?(\widehat{t} \ge 0?0 : (\widehat{t}/t)) \\ :(\widehat{t} \le 0?0 : (\widehat{t}/t))$$

During type checking, rule (T-ODOT) checks the following constraint for the branch condition $\eta \leq t \Leftrightarrow \eta + \eta \cdot \mathbb{d} \leq t + \widehat{t}$, which can be discharged by a SMT solver. Hence, the algorithm is transformed to the program at the bottom of Figure 13. By the fact that the newly added random source and typing rules satisfies Lemma 2, the privacy cost of this subtle example is provably bounded by the transformed cost formula in the transformed program [7].

7. Related Work

Type systems for differential privacy Fuzz (Reed and Pierce 2010) and its successor DFuzz (Gaboardi et al. 2013) reason about the *sensitivity* (i.e., how much does a function magnify distances between inputs) of a program. DFuzz combines linear indexed types and lightweight dependent types to allow rich sensitivity analysis. However, those systems rely on (without verify) external mechanisms (e.g., Laplace mechanism, Sparse Vector method) as trusted black boxes to release final query answers, without verifying those black boxes. LightDP, on the other hand, verifies sophisticated privacy-preserving mechanisms that releases those final answers. Sensitivity inference (D'Antoni et al. 2013) was proposed in the context of Fuzz. While sensitivity inference shares the same goal of minimizing type annotation and it also uses SMT solvers, the very different type system in LightDP brings unique challenges (Section 5.1) that do not present in Fuzz.

HOARe^2 (Barthe et al. 2015) and its extension PrivInfer (Barthe et al. 2016a) have the ability to relate a pair of expressions via relational assertions that appear as refinements in types. Hence, they can verify mechanisms that privately release final query answers as well as private Bayesian inference algorithms. However, HOARe^2 and PrivInfer incur heavy annotation burden on programmers. Moreover, they can not deal with privacy-preserving algorithms that go beyond the composition theorem (e.g., the Sparse Vector method).

Program logic for differential privacy Probabilistic relational program logic (Barthe et al. 2012, 2013; Barthe and Olmedo 2013; Barthe et al. 2016c,b) use custom relational logics to verify differential privacy. These systems have successfully verified privacy for many advanced examples. However, only the very recent work by Barthe et al. (2016c,b) can verify the Sparse Vector method. While these logics are expressive enough to prove (ϵ, δ) privacy, the main difficulty with these approaches is that they use custom and complex logics that incurs steep learning curve and heavy annotation burden. Moreover, ad hoc rules for loops are needed for many advanced examples.

The work by Barthe et al. (2014) transforms a probabilistic relational program to a nondeterministic program, where standard Hoare logic can be used to reason about privacy. However, the fundamental difference between that work and LightDP is that the former cannot verify sophisticated algorithms where the composition theorem falls short, since it lacks the power to express subtle dependency between privacy cost and memory state. Moreover, beneath the surface, that work and LightDP are built on very different principals and proof techniques. Further, their approach requires heavier annotation burden since both relational and functional (e.g., bounding privacy cost) properties are reasoned about in the transformed program, while the former is completely and automatically handled by the type system of LightDP.

The notion of aligning randomness has been used in the recent coupling method (Barthe et al. 2016c,b). While the coupling method is capable of proving (ϵ, δ) privacy and it does not require the injective assumption on the alignment, the cost of doing so is the steep learning curve and heavy annotation burden. Technically, the coupling method reasons about privacy for each possible output (or a set of outputs), while the alignment-based theory used in this paper aligns two program executions that will produce the same results. The theory in this paper gives a simple proof, a light-weight type system, and clear insight behind the type system.

Other language-based methods for differential privacy Several dynamic tools exist for enforcing differential privacy. PINQ (McSherry 2009) tracks (at runtime) the privacy budget consumption, and terminates the computation when the privacy budget is exhausted. Airavat (Roy et al. 2010) is a MapReduce-based system with a runtime monitor that enforces privacy policies controlled by data providers. Recent work by Ebadi et al. (2015) proposed Personalised Differential Privacy (PDP), where each individual has its own personal privacy level and a dynamic system that implements PDP. There are also methods based on computing bisimulations families for probabilistic automata (Tschantz et al. 2011; Xu et al. 2014). However, none of these techniques has the expressive power to provide a tight privacy cost bound for sophisticated privacy-preserving algorithms.

8. Conclusions and Future Work

The increased usage and deployment of differentially private algorithms underscores the need for formal verification methods to ensure that personal information is not leaked due to mistakes or carelessness. The ability to verify subtle algorithms should be coupled with the ability to infer most of the proofs of correctness to reduce the programmer burden during the development and subsequent maintenance of a privacy-preserving code base.

In this paper, we present a language with a lightweight type system that allows us to separate privacy computation from the alignment of random variables in hypothetical executions under related databases. Thus enabling inference and search for proofs with the minimal privacy costs.

These techniques allow us to verify (with much fewer annotations) algorithms that were out of reach of the state of the art until recently. However, additional extensions are possible. The first challenge is to extend these methods to algorithms that use hidden private state to reduce privacy costs. One example is the noisy max algorithm that adds noise to each query and returns the index of the query with the largest noisy answer (although all noisy answers are used in this computation, the fact that their values are kept secret allows more refined reasoning to replace the composition theorem). The second challenge is verifying subtle algorithms such as PrivTree (Zhang et al. 2016), in which intermediate privacy costs depend on the data (hence cannot be released) but their sum can be bounded in a data-independent way. This is another case where the composition theorem can fail since it requires data-independent privacy costs. Lastly, LightDP currently only verifies ϵ-privacy, which has a nice point-wise property. We leave extending LightDP to (ϵ, δ)-privacy as future work.

Acknowledgments

We thank Adam Smith, our shepherd Marco Gaboardi and anonymous reviewers for their helpful suggestions. This work was supported by NSF grants CNS-1228669 and CCF-1566411.

[7] We note that without LightDP, the precise calculation of privacy cost is very difficult and error-prone. To show that the randomness alignment cancels out the difference in the private-data-dependent probability t, we need to analyze four cases. When outputting `true` and $\widehat{t} \geq 0$, the related execution must output `true` as well ($\eta \leq t \wedge \widehat{t} \geq 0 \Rightarrow \eta \leq t + \widehat{t}$). When outputting `true` and $\widehat{t} < 0$, this alignment maps η to $\eta(\mathbb{d} + 1) = \eta((t + \widehat{t})/t)$. Hence, `true` is the output in the related execution ($\eta \leq t \Rightarrow \eta(t+\widehat{t})/t \leq t+\widehat{t}$). Similar reasoning applies to the case outputting `false` too. Moreover, connecting this alignment to ϵ-privacy require is even more daunting by a paper-and-pencil proof.

References

A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *FPLCA*, pages 31–41, 1993.

G. Barthe and F. Olmedo. Beyond differential privacy: Composition theorems and relational logic for f-divergences between probabilistic programs. In *ICALP*, pages 49–60, 2013.

G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–110, 2012.

G. Barthe, G. Danezis, B. Grégoire, C. Kunz, and S. Zanella-Béguelin. Verified computational differential privacy with applications to smart metering. In *2013 IEEE 26th Computer Security Foundations Symposium*, pages 287–301, 2013.

G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, C. Kunz, and P. Y. Strub. Proving differential privacy in hoare logic. In *2014 IEEE 27th Computer Security Foundations Symposium*, pages 411–424, 2014.

G. Barthe, M. Gaboardi, E. J. G. Arias, J. Hsu, A. Roth, and P. Strub. Higher-order approximate relational refinement types for mechanism design and differential privacy. In *POPL*, 2015.

G. Barthe, G. P. Farina, M. Gaboardi, E. J. G. Arias, A. Gordon, J. Hsu, and P.-Y. Strub. Differentially private bayesian programming. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 68–79, 2016a.

G. Barthe, N. Fong, M. Gaboardi, B. Grégoire, J. Hsu, and P.-Y. Strub. Advanced probabilistic couplings for differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 55–67, 2016b.

G. Barthe, M. Gaboardi, B. Gregoire, J. Hsu, and P.-Y. Strub. Proving differential privacy via probabilistic couplings. In *IEEE Symposium on Logic in Computer Science (LICS)*, 2016c. To apprear.

N. Bjørner and A.-D. Phan. νZ — maximal satisfaction with Z3. In T. Kutsia and A. Voronkov, editors, *6th International Symposium on Symbolic Computation in Software Science (SCSS)*, volume 30 of *EPiC Series in Computing*, pages 1–9, 2014.

N. Bjørner, A.-D. Phan, and L. Fleckenstein. *νZ — An Optimizing SMT Solver*, pages 194–199. 2015.

H. Chan, E. Shi, and D. Song. Private and continual release of statistics. *ACM Transactions on Information and System Security*, 14(3), 2011.

Y. Chen and A. Machanavajjhala. On the privacy properties of variants on the sparse vector technique. http://arxiv.org/abs/1508.07306, 2015.

L. D'Antoni, M. Gaboardi, E. J. Gallego Arias, A. Haeberlen, and B. Pierce. Sensitivity analysis using type-based constraints. In *Proceedings of the 1st Annual Workshop on Functional Programming Concepts in Domain-specific Languages*, pages 43–50, 2013.

L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

C. Dwork and A. Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3–4):211–407, 2014. ISSN 1551-305X. doi: 10.1561/0400000042.

C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. In *EUROCRYPT*, pages 486–503, 2006a.

C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, 2006b.

H. Ebadi, D. Sands, and G. Schneider. Differential privacy: Now it's getting personal. In *POPL*, 2015.

U. Erlingsson, V. Pihur, and A. Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, 2014.

M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 357–370, 2013.

A. Greenberg. Apple's 'differential privacy' is about collecting your data – but not *Your* data. Wired, `https://www.wired.com/2016/06/apples-differential-privacy-collecting-data/`, 2016.

C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1–3):189–224, 2004.

D. Kifer and A. Machanavajjhala. Pufferfish: A framework for mathematical privacy definitions. *ACM Trans. Database Syst.*, 39(1):3:1–3:36, 2014.

D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328 – 350, 1981.

K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370, 2010.

M. Lyu, D. Su, and N. Li. Understanding the sparse vector technique for differential privacy. https://arxiv.org/abs/1603.01699, 2016.

A. Machanavajjhala, D. Kifer, J. Abowd, J. Gehrke, and L. Vilhuber. Privacy: From theory to practice on the map. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pages 277–286, 2008.

P. Martin-Löf. Intuitionistic type theory. *Naples: Bibliopolis*, 76, 1984.

F. McSherry and K. Talwar. Mechanism design via differential privacy. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, pages 94–103, 2007.

F. D. McSherry. Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 19–30, 2009.

P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler. Gupt: Privacy preserving data analysis made easy. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2012.

J. Reed and B. 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*, ICFP '10, pages 157–168, 2010.

A. Roth. The sparse vector technique. `http://www.cis.upenn.edu/~aaroth/courses/slides/Lecture11.pdf`, 2011.

I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: Security and privacy for MapReduce. In *NSDI*, 2010.

M. C. Tschantz, D. Kaynar, and A. Datta. Formal verification of differential privacy for interactive systems (extended abstract). *Electron. Notes Theor. Comput. Sci.*, 276:61–79, Sept. 2011.

M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987.

L. Xu, K. Chatzikokolakis, and H. Lin. *Metrics for Differential Privacy in Concurrent Systems*, pages 199–215. 2014.

D. Zhang and D. Kifer. LightDP: Towards automating differential privacy proofs. *CoRR*, abs/1607.08228, 2016. URL `http://arxiv.org/abs/1607.08228`.

D. Zhang and A. C. Myers. Toward general diagnosis of static errors. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 569–581, Jan. 2014.

J. Zhang, X. Xiao, and X. Xie. Privtree: A differentially private algorithm for hierarchical decompositions. In *SIGMOD*, 2016.