AutoPriv: 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 type systems lacks expressive power for some sophisticated algorithms.

In this paper, we present AutoPriv, a simple imperative language that strikes a better balance between expressive power and usefulness. The core of AutoPriv 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 AutoPriv infers most of the proof details, and even searches for the proof with minimal privacy cost when multiple proofs exist. We show that AutoPriv verifies sophisticated algorithms with little manual effort.

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, under very general conditions (Kifer and Machanavajjhala 2014), provably limit the ability of attackers to infer individual-level sensitive information.

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 of the most recent cases.

Currently, there are two strategies for combating this trend. The first is the use of programming platforms (McSherry 2009; Mohan et al. 2013; 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; Gaboridi et al. 2013; Barthe et al. 2012, 2014, 2016). 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. (2013). 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 AutoPriv, a language for developing provably privacy-preserving algorithms. The goal of AutoPriv 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. 2016), was beyond the capabilities of verification tools). For example, we show that the Sparse Vector method can be verified in AutoPriv with just two lines of annotation from the programmer.

AutoPriv 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 depended 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; Gaboridi et al. 2013; Barthe et al. 2012, 2014) based on the composition theorem (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, AutoPriv has an inference engine that reduces the already low type-annotation burden on the programmers. The inference engine not only fills in missing proof details; based on MaxSMT theory, it even searches for the optimal proof that minimizes privacy cost. For example, with only two lines of annotation...
tions, AutoPriv confirms that the proof in [Dwork and Roth 2014] indeed provides the minimal privacy cost.

To summarize, this paper makes the following contributions:
1. AutoPriv, a new imperative language for verifying sophisticated privacy-preserving algorithms (Section 3).
2. expressive static annotations incorporating dependent types, enabling precise tracking of privacy costs (Section 3.3).
3. a formal proof that the AutoPriv type system soundly differential privacy costs, and new proof techniques involved in the soundness proof (Section 4).
4. an inference engine that automatically fills in missing details, and further, minimizes provable privacy cost when multiple proofs exist (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 as a set over A, written Dist(A), as the set of functions \( \mu : A \rightarrow [0, 1] \), such that \( \sum_{a \in A} \mu(a) \leq 1 \). When applied to an event \( E \subseteq A \), we define \( \mu(E) \triangleq \sum_{a \in E} \mu(a) \). Notice that we do not require \( \sum_{a \in A} \mu(a) = 1 \), a special case when \( \mu \) is a distribution, since sub-distribution gives rise to an elegant semantics for programs that may not terminate [Kozen 1981].

Given a distribution \( \mu \in \text{Dist}(A) \), its support is defined as \( \text{support}(\mu) \triangleq \{ a \mid \mu(a) > 0 \} \). We use \( \mathbb{I}_a \) to represent the degenerate distribution \( \mu(a) = 1 \) 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:

\[
\begin{align*}
\text{unit} : A \rightarrow \text{Dist}(A) & \triangleq \lambda a. \mathbb{I}_a \\
\text{bind} : \text{Dist}(A) \rightarrow (A \rightarrow \text{Dist}(B)) \rightarrow \text{Dist}(B) & \triangleq \lambda \mu. \lambda b. \sum_{a \in A} (f a b) \times \mu(a))
\end{align*}
\]

That is, unit takes an element in \( A \) and returns the Dirac distribution where all mass is assigned to \( a \); bind takes \( \mu \), a distribution on \( A \), and \( f \), a mapping from \( A \) to distributions on \( B \), and returns the corresponding marginal distribution on \( B \). This Monadic view will avoid cluttered definitions and proofs when probabilistic programs are involved (Section 3.3).

2.2 Differential Privacy

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

Definition 1 (Differential privacy). Let \( \epsilon, \delta > 0 \). A probabilistic computation \( M : A \rightarrow \text{Dist}(B) \) is \((\epsilon, \delta)\)-differential 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 \( a_1, a_2 \in A \) and every output subset \( E \subseteq B \), we have

\[
P(M(a_1) \in E) \leq \exp(\epsilon) P(M(a_2) \in E) + \delta
\]

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

2.3 The Sparse Vector method

The goal of formal methods for verifying \((\epsilon, \delta)\)-differential privacy is to provide an upper bound on the privacy costs \( \epsilon \) and \( \delta \) of a program. Typically, users will have a fixed privacy budget \( \epsilon' \) and \( \delta' \) and can only run programs whose provable private cost \( \epsilon, \delta \) does not exceed the budget: \( \epsilon \leq \epsilon' \) and \( \delta \leq \delta' \). 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. 2016], most existing formal methods rely on the composition theorem [McSherry 2009], which (in the case of \( \epsilon \)-differential privacy) essentially treats a program as a series of modules, each with a privacy cost \( \epsilon_i \) and a total privacy cost bounded by \( \sum \epsilon_i \). 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 2013; 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 counting queries \( q_1, q_2, q_3, \ldots \) (e.g., how many patients in the data have Cancer, how many patients contracted 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, \( \text{Lap}(4N/\epsilon) \) draws one sample from the Laplace distribution with mean zero and a scale factor \( (4N/\epsilon) \). If the noise query answer is above the threshold, it outputs \text{true} for that query and otherwise, it outputs \text{false}. The key to this algorithm is the deep observation that once noise has been added to the threshold, queries for which we output \text{true} have a privacy cost (so we can answer at most \( N \) of them, where \( N \) is a parameter). However, outputting \text{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 \( \epsilon \), 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 \( \text{Lap}(4N/\epsilon) \) noise is added to a query answer). Since the number of iterations are not a priori bounded, the composition theorem could not prove that the algorithm satisfies \( (\epsilon^*, 0) \)-differential privacy for any finite \( \epsilon^* \). More advanced methods are needed to prove that it satisfies \((\epsilon, 0)\)-differential privacy.

**Informal proof and sample runthrough** Proofs of correctness (of this and other variants) can be found in (Dwork and Roth 2014, Chen and Machanavajjhala 2013, Lyu et al 2016). Here we provide an informal proof 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 \( \text{true} \)). Consider the following two databases \( D_1, D_2 \) that differ on one record, and their corresponding query answers:

\[
\begin{align*}
D_1 : & \quad q[0] = 2, \quad q[1] = 3, \quad q[2] = 5 \\
D_2 : & \quad q[0] = 3, \quad q[1] = 3, \quad q[2] = 4
\end{align*}
\]

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 \( \beta^{(2)} = 2, \beta^{(1)} = 0, \beta^{(2)} = 0 \), respectively. Thus the noisy threshold is \( T = 5 \) and the noisy query answers are \( q[0] + \beta^{(2)} = 4, q[1] + \beta^{(1)} = 3, q[2] + \beta^{(2)} = 5 \) and so the algorithm outputs the sequence: \( \text{false, false, true} \).

For any output sequence \( \omega \), we need to show \( P(M(D_1) = \omega) \leq e^\epsilon P(M(D_2) = \omega) \) for all possible outputs \( \omega \) and databases \( D_1, D_2 \) that differ on one record. For our particular example, we need to show that \( P(M(D_1) = \text{false, false, true}) \leq e^\epsilon P(M(D_2) = \text{false, false, true}) \). We proceed in two steps.

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

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

**Counting privacy cost** For each output \( \omega \), let \( A \) be the set of possible random bits that cause execution under \( D_1 \) to produce \( \omega \). \( B \) be the possible randomness we can get by applying the injective mapping to \( A \), and \( C \) be the set of randomness that is not in the range of the mapping, but nevertheless causes execution under \( D_2 \) to produce the output \( \omega \) as well. 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, the proof finishes by showing:

\[
P(A) \leq e^\epsilon e^{\epsilon^*} P(B) = e^\epsilon P(B) \leq e^\epsilon (P(B) + P(C))
\]

where the \( e^\epsilon \) factor results from using a threshold value that is 1 larger, while the \( e^{\epsilon^*} \) factor results from adding 2 to the noise for query \( q[2] \) (and then we note that the parameter \( N \) was set to 1). Notice that no privacy cost is paid for queries \( q[0] \) and \( q[1] \), since the same noise is added under \( D_1 \) and \( D_2 \).

**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 \text{true}. Hence, the cost calculation needs to consider program states.

2. Aligning randomness: finding an injective mapping from the randomness under \( D_1 \) to that under \( D_2 \) such that outputting \( \omega \) under \( D_1 \) entails outputting \( \omega \) under \( D_2 \) is the most intriguing piece in the proof. However, coming up with a correct mapping, 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 cost. Since a tighter privacy cost bound allows an 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. (2016), all existing formal methods (e.g., (Reed and Pierce 2010, Gaboardi et al. 2013, Barthe et al. 2012, 2014)) solely rely on the composition theorem, hence fail to prove that the Sparse Vector method satisfies \( \epsilon^* \)-privacy for any \( \epsilon^* \). Recent work by Barthe et al. (2016) verifies the Sparse Vector method when one \text{true} output is allowed (i.e., \( N = 1 \)), but its customized relational Hoare logic incurs heavy annotation burden, including the randomness alignment. Moreover, the work by Barthe et al. (2016) cannot search for the tightest cost bound.

### 2.4 Our approach

To tackle the challenges above, we propose AutoPriv, 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 AutoPriv in this section, and detail all components in the rest of this paper.

**Relational reasoning** The core of AutoPriv is a novel lightweight 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 \( x \) for \( x \) in AutoPriv has the form of \( \text{B}_x \), meaning that \( x \) holds a value of basic type \( B \) (e.g., int, real, bool), and \( v^{(1)} + c = v^{(2)} \). Required type annotations for the Sparse Vector method is shown in grey in Figure 1. Note that for brevity, we write \( \text{num} \) for numeric base types (e.g., int, real). Hereafter, we refer to the \( c \) counterpart as the distance.

---

1 This is possible since we do not assume the mapping to be a bijection.
In the simplest case, the distance is a constant. For example, the input threshold $T$ has a type $\text{num}$, 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 $\eta_1$ captures one randomness alignment in the informal proof above: we enforce a distance of 1 for $\eta_1$ (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, AutoPriv supports dependent types. For instance, consider the distance of $\eta_2$: $q[i] + \eta_2 \geq T + 2$ $\vdash 0$. This annotation specifies an injective mapping that maps noise $v$ to $v+2$ when the value of $q[i]+\eta_2$ 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, such 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, AutoPriv uses a distinguished distance $\epsilon$ as a shorthand for the standard Sigma type (e.g., $\text{num} \equiv \Sigma_{x: \text{num}} x$). In other words, each type of $\text{num}$, (e.g., each query in the list $q$) can be interpreted as a pair of the form $(x : \text{num}, y : \text{num})$, where the first component specifies the distance of the second component. Note that by language design, the first component is not accessible in the privacy-preserving algorithm. However, the type system may reason about and manipulate the first component via a distinguished operation $\diamond$. For instance, the precondition in the running example states the adjacent assumption on databases: for each query answer $q[i]$ in $q$, its distance ($\tilde{q}[i]$) is bounded by $\pm 1$. 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 list $\text{num}$, meaning that each element in the list has type $\text{num}$. Hence, the invariant maintained by the type system ensures that both 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 $v_1$, and 2) probabilistic instructions are replaced by a new nondeterministic instruction $\text{havoc } \eta$, which semantically sets variable $\eta$ 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) $v_1$ is always bounded by some constant $\epsilon$ in the transformed program, then the original program being verified is $\epsilon$-private.

Notice that for the second property (the problem of bounding $v_1$), any off-the-shelf verification tool for functional correctness can be utilized. For instance, the program above with the desired postcondition $v_1 \leq \epsilon$ 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 provides a light-weight yet powerful formal method for differential privacy. The annotation burden is much reduced compared with [Barthe et al. 2016]. However, providing the correct and optimal type annotations (especially for random variables $\eta_1$ and $\eta_2$) is still subtle for the programmer.

Although it is folklore that type inference in face of dependent types can be daunting, AutoPriv is equipped with an inference engine that not only infers correct annotations, but also enables finding annotations that minimizes 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 $\eta_1, \eta_2$ are assigned with a distance $\alpha$, $\beta$ is assigned with a distance expression $\eta_2 = T + 2$ $\beta := \gamma$, where $\alpha, \beta, \gamma$ are variables to be inferred, subjecting 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 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 type variables, the type-guided transformation as sketched above generates target program where variables to be inferred (i.e., $\alpha, \beta, \gamma$) are used in the calculation of privacy cost. The difference is that $v_1$ increments by $(\alpha \epsilon) / 2$ at line 2 and increments by $(\eta_1 + \eta_2 \geq T / \beta := \gamma) \times \epsilon / 2N$ at line 6 in Figure 2. By Hoare logic, we can easily bound the privacy cost to be $(\alpha) / 2 + \beta \epsilon/4 + c_2 \times \gamma \epsilon / 4N$.

Putting together, finding the optimal proof is equivalent to a MaxSMT problem: min $(2\alpha + \beta + c_2 \times \gamma / N)$, given that constraints generated in type checking are satisfiable. Using an existing MaxSMT solver, $\mu Z$ [Björnstrom and Phani 2014; Björnstrom et al. 2018], 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 formal proof [Dwork and Roth 2014].

3. AutoPriv: A language for algorithm design

We first introduce a simple imperative language, AutoPriv, 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. AutoPriv0 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 AutoPriv0 follows a modular design where new randomness expression can be added easily, we only consider the most interesting random expression, \( \text{Lap } r \), for now. Semantically, \( \text{Lap } r \) draws one sample from the Laplace distribution, with mean zero and a scale factor \( r \). We will discuss different random expressions in Section 6.3.

Each random expression \( g \) can be assigned to an random variable \( \eta \), written as \( g \triangleq \eta \). We distinguish random variables \( \eta \) from normal variables \( \text{Var} \) for technical reasons explained in Section 6.3. 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 \( \eta \) 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, AutoPriv0 enables more precise reasoning via a built-in list type with Lisp-style operations: \( \text{cons } e_1 \ e_2 \) appends the element \( e_1 \) to a list \( e_2 \); \( e \ e_1 \) gets the \( e \)-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 \( \tau \) has the form of \( B \ e \). Here, \( B \) is a base type, such as \( \text{num} \) (numeric type), \( \text{bool} \) (Boolean), or an application of a type constructor (e.g., \( \text{list} \)). A type \( d \) is a side-effect-free expression that (semantically) specifies the exact distance of the values stored in a variable in two related executions. In particular, a distance expression is any numeric expression the language, as specified in Figure 3 where \( d_1 \circ d_2 \circ d_3 : d_4 \) evaluates to \( d_4 \) when the comparison evaluates to \( \text{true} \), and \( d_4 \) otherwise.

Since non-numeric types \( \text{bool} \) and \( \text{list} \) cannot be associated with any numeric distance, type \( \text{bool} \) and \( \text{list} \) are syntactic sugars for \( \text{bool}_0 \) and \( \text{list}_0 \) respectively. Notice that elements in a list of type \( \text{list}_0 \) (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 \( \tau \). The subscript \( 0 \) here simply restricts the list size in two related executions.

**Star type** AutoPriv0 also supports sum types, written as \( B \_s \), a syntactic sugar of a Sigma type. More specifically, a variable \( x \) with type \( \text{num} \_s \) is desugared as \( x : \Sigma_{\text{num}_s} \text{num}_s \), where \( x \) 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 6).

The parameter \( q \) in Figure 4 is one example where the star type is useful. Moreover, the star type enables the reasoning of dependencies that cannot be captured otherwise by a distance expression.

**Example: Partial Sum**

```
function PARTIALSUM(num0 : e, b, size, list num : q)
returns num0; out
precondition \forall i \geq 0. q[i] \leq b ∧ \forall i \geq 0. q[i] \geq 0 \Rightarrow (q[i] > q[j]) = 0
1 num_0 : sum; num_0 : i; num_0 : q : η
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 \( ε \)-private algorithm for summing over a query list.

Consider the Partial Sum algorithm in Figure 4. This \( ε \)-private algorithm implements an immediate solution of answering the sum of a query list in a privacy preserving manner. It aggregates the accurate partial sum in a loop, and releases a noisy sum using the Laplace mechanism. The precondition specifies the adjacent assumption: at most one query answer may differ by at most \( b \).

In this algorithm, the distance of variable \( \text{num}_0 \) changes in each iteration. Hence, the accurate type for \( \text{num}_0 \) is \( \text{num}_0^{b_{m_{i+6}} \eta_{\text{num}_0}} \). However, with the goal of keeping type system as light-weight as possible, we assign \( \text{num}_0 \) to a star type. The type system will reason about and manipulate distance component \( \text{num}_0 \) in a sound way (Section 3.2).

**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 memories states where each memory state \( m \in \mathcal{M} \) is an assignment of all (normal and random) variables \( \text{Var} \cup \text{H} \) to values. First, an expression \( e \) of base type \( B \) is interpreted as a function \( [e] : m \rightarrow \text{Dist}(B) \), where \( \text{Dist}(B) \) represents the set of values belonging to the base type \( B \). We omit expression semantics since it is completely standard.

A random expression \( g \) is interpreted as a distribution on real values. Hence, \( [g] : \text{Dist}(\text{num}) \). Moreover, a command \( c \) is interpreted as a function \( [c] : \mathcal{M} \rightarrow \text{Dist}(\mathcal{M}) \).

**Figure 5.** AutoPriv0: language semantics.

We use this trivial algorithm here for its simplicity. We will analyze a more sophisticated version in Section 6.
directly to a semantics given by [Kozel (1981)], which interprets
programs as continuous linear operators on measures.

Finally, we assume all programs have the form \((c; \text{return } e)\) where \(c\) does not contain return statements. An AutoPriv\(\text{P} \varnothing\) program is interpreted as a function \(m \rightarrow \text{Dist}[\mathcal{B}]\), 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 \(\Gamma\) that tracks the type of each variable (including random variable). For now, we assume type annotation is provided for each variable (i.e., \(\text{dom}(\Gamma) = \varnothing \cup \mathcal{H}\)), 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 \(\Psi\) (e.g., the precondition in Figure 1), typing rules do not propagate \(\Psi\) for brevity. We also write \(\Gamma(x) = \varnothing\) for \(\forall x\). \(\Gamma(x) = \mathcal{B}\) 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 \(\tau\) under the environment \(\Gamma\). Rule (T-OP\(\text{STAR}\)) precisely tracks the distances of linear operations (e.g., \(+\) and \(-\)), while rule (T-OT\(\text{IMES}\)) makes a conservative assumption that other numerical operations take identical parameters. It is completely possible to refine rule (T-OT\(\text{IMES}\)) (e.g., by following the sensitivity analysis proposed by [Reed and Pierced 2016; Gaboardi et al. 2013]) to improve precision, however, we leave that as future work since it is largely orthogonal.

Rule (T-V\(\text{AR\(\text{STAR}\)}\)) applies when variable \(x\) has a star type. This rule unpacks the corresponding pair with Sigma type and makes \(\hat{x}\) explicit in the type system.

The most interesting and novel rule is (T-OD\(\text{OT}\)). It type-checks a comparison of two real expressions by generating a constraint:

\[
\Psi \Rightarrow (e_1 \circ e_2 \Leftrightarrow (e_1 + d_1) \circ (e_2 + d_2))
\]

Intuitively, this constraint requires that in two related executions, the Boolean value of \(e_1 \circ e_2\) must be identical since the distances of \(e_1\) and \(e_2\) are specified by \(d_1\) and \(d_2\), respectively.

For example, consider the branch condition \(q[i] + q_2 \geq T\) in Figure 1. Rule (T-OD\(\text{OT}\)) first checks types for subexpressions:

\[
\Gamma \vdash q[i] + q_2 : \text{num}_1
\]

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

\[
\forall q, q_i, q_2, T \in \mathbb{R}. (1 \leq q_i \leq 1) \Rightarrow q + q_2 \geq T \Rightarrow q + q_2 + q_i + (q + q_2 \geq T ? 2 : 0) \geq T + 1
\]

This proof obligation captures a subtle yet important property of the Sparse Vector method: given the randomness alignment as specified by \(\Gamma\), 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 rule for a command has the form of \(\Gamma \vdash c \rightarrow \hat{c}'\), where \(c\) is the original program being verified, and \(\hat{c}'\) is the transformed program in the target language defined in Figure 4. The target language is mostly identical to the original one, except for two significant differences: 1) the target language involves a distinguished variables \(\mathcal{V}_c\) to explicitly track the privacy cost in the original program; 2) the target language removes probabilistic expressions, and introduces a new nondeterministic command (\text{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 \(c] : \mathcal{M} \rightarrow \mathcal{P}(\mathcal{M})\). For example, the semantics of the \text{havoc} command is defined as

```
Figure 6. Typing rules. \(\Psi\) is an invariant that holds throughout program execution.
```
executions remains unchanged. Hence, the type invariant is broken. Morphic type to the random source
\( v \) \( \eta \) in line 7. In other words, for any distance of \( \epsilon \), Laplace distribution.

For a possible pair of initial and final memories \( \mu_1, \mu_2 \in \text{Dist} (M) \), written \( \Delta \Gamma_i (\mu_1, \mu_2) \), is defined as:
\[
\Delta \Gamma_i (\mu_1, \mu_2) \triangleq \max_{S \subseteq c} \exp (\epsilon/2) (\mu_1 (S) - \exp (\epsilon/2) (\Gamma (S))
\]

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

The soundness theorem connects the “privacy cost” of the probabilistic program to the distinguished variable \( v \), in the transformed nonprobabilistic program. In order to formalize the connection, we first extend memory in the source language to include \( v \).

Definition 4. For any memory \( m \) and constant \( \epsilon \), there is an extension of \( m \), written \( m \uplus (\epsilon) \), so that
\[
\forall x \in \text{dom} (m). \ m \uplus (\epsilon) (x) = m(x) \land m \uplus (\epsilon) (v_\eta) = \epsilon
\]

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

Lemma 1 (Faithfulness).
\[
\forall m, m', c, c', \Gamma. \ \Gamma \vdash c \rightarrow c' \Rightarrow \ \llbracket c \rrbracket [m] (m') \neq 0 \Leftrightarrow \exists \epsilon. m' \uplus (\epsilon) (c) = \llbracket c' \rrbracket [m0] (0)
\]

Proof. By structural induction on \( c \). □

For a possible pair of initial and final memories \( m \) and \( m' \) when executing the original program, we identify a set of possible \( v \) values, so that in the corresponding executions of \( c' \), the initial and final memories are extensions of \( m \) and \( m' \) respectively:

4. Soundness

The type system in Section 3 enforces a fundamental property: if \( \Gamma \vdash c \rightarrow c' \) and \( v_\eta \) in \( c' \) is bounded by some constant \( \epsilon \), then the original program being verified is \( \epsilon \)-private.

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

Definition 2 (\( \Gamma \)-Relation). Two memories \( m_1 \) and \( m_2 \) are related by a typing environment \( \Gamma \), written \( m_1 \Gamma m_2 \), iff
\[
\forall x \in \text{Var}. \ m_1(x) + \llbracket d \rrbracket [m_1] = m_2(x), \ \text{where} \ \Gamma \vdash x : B_{\Delta}
\]

Note that since \( \Gamma (x) \) might be a dependent type, the definition evaluates the distance of \( x(d) \) under \( m_1 \).

Moreover, we assume that \( \Gamma \) is an injective mapping. Hence, we also write \( \Gamma (m) \) to represent the unique \( m' \) such that \( m \Gamma m' \). Moreover, given a set of memories \( S \subseteq M \), we define \( \Gamma S \triangleq \{ \Gamma (m) \mid m \in S \} \).

For differential privacy, we are interested in the relationship of two memory distributions. Given a typing environment \( \Gamma \) and constant \( \epsilon \), we define a \( (\Gamma, \epsilon) \) distance, written \( \Delta \Gamma_i \), of two memory distributions:

\[
\Delta \Gamma_i (\mu_1, \mu_2) \triangleq \max_{S \subseteq c} \exp (\epsilon/2) (\mu_1 (S) - \exp (\epsilon/2) (\Gamma (S))
\]

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 \( \text{num}_a \), 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 \( x \in \text{Var} \) such that \( \Gamma (x) = B_{\Delta} \), all free variables in \( d \) are immutable. For example, each normal variables in Figure 1 has a constant distance in its type. Second, a random variable \( \eta \in H \) may depend on mutable variables. However, we assume that it has only one use other than the definition, and the definition of \( \eta \) is adjacent to its use. Hence, each variable that \( \eta \) depends on appears immutable between \( \eta \)'s definition and use.
Definition 5. 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 $c' | m_0 \nRightarrow m'$, is defined as follows:

$$c' | m_0 \nRightarrow m' \triangleq \{ m | m \uplus (e') \in [c'] | m_0 \nRightarrow (0) \land m = m' \}$$

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

Since $(c' | m_0 \nRightarrow m')$ by definition is a set of values of $c'$, we write $\max(c' | m_0 \nRightarrow m')$ for the maximum cost. The next lemma enables precise reasoning of privacy cost w.r.t. a pair of initial and final memories:

Lemma 2 (Point-Wise Soundness).\[\forall c, c', m_1, m_2, m, \Gamma \vdash c \rightarrow c' \land m_1 \equiv \Gamma m_2, \text{ we have } \] $$\|c\|_{m_1}(m) \leq \exp(\max(c' | m_0 \nRightarrow m'))[c]_{m_2}($$$\Gamma(m))$$

The full proof of Lemma 2 is available in the appendix. 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:

Theorem 1 (Soundness).\[\forall c, c', m_1, m_2, m, \Gamma \vdash c \rightarrow c' \land m_1 \equiv \Gamma m_2 \land c' \leq c
\]

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 $c' \leq c$ holds for some $c$. For example, we have formally proved that the transformed program in Figure 1 satisfies a postcondition $\forall c \leq c'$. We use a one line of annotation (the grey line in Figure 1) using the Dafny tool (Leino 2010).

Theorem 2 (Privacy).\[\forall \Gamma, c, c', x, c, \Gamma \vdash (c; \text{return } x) \rightarrow (c'; \text{return } x) \text{ then } \] $$\Gamma(x) = B_0 \land c' \leq c \Rightarrow c \text{ is } c\text{-private}$$

Proof. Proof is available in the appendix.

5. AutoPriv: differential-privacy proof inference

We have so far presented an explicitly typed language AutoPriv0. 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 AutoPriv0 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.

Next, we assume all base types are available, and focus on the inference of the distance counterpart. For brevity, we write $\Gamma(x) = c$ instead of $\Gamma(x) = B_c$. We use DefVar to represent the set of variables whose distances are given by the programmer.

To enable type inference, we extend AutoPriv0 with distance variables such as $\alpha, \beta, \gamma$ (shown in Figure 3). Initially, the typing environment associates each variable in DefVar 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 2003)), it is straightforward to interpret the typing rules in Figure 4 as a (naive) inference algorithm. To see how, consider two assignments $(x := 0; y := i)$, 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 $\exists \alpha, \beta, \alpha = 0 \land \beta = \alpha$). It is easy to check that $\alpha = 0 \land \beta = 0$ is a solution. Hence, the inferred distances are $\Gamma(x) = 0, \Gamma(y) = 0$. However, this naive inference algorithm fails 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 $\alpha = \alpha, \beta = \alpha$. These constraints are unsatisfiable, since the value of $\alpha$ is unknown at compile time. Nevertheless, the powerful type system of AutoPriv0 still allows formal verification of this example by assigning $\alpha$ 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 4. Then, if $x \in \text{DefVar}$ (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:

$$\text{refine}(\Gamma, x, c) \triangleq \begin{cases} \Gamma[c/\alpha] & \text{if } \Gamma(x) = \alpha \\ \Gamma & \text{if } \Gamma(x) \neq \alpha \land (\Gamma(x) = c) \\ \Gamma[x \mapsto *] & \text{otherwise} \end{cases}$$

Here, the auxiliary function refine takes an initial environment $\Gamma$, and returns a new environment where type of variable $x$ is refined.
to $d$. This function replaces all occurrences of $\alpha$ in $\Gamma$ to $d$ when $\Gamma(x) = \alpha$ is a variable to be inferred. 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 3 again. At line 4, num's distance is refined to 0. Then at line 6, its distance is refined to $*$, since we cannot statically check that $0 = \hat{q}[i]$ is valid.

**Refinement rules for expressions**

\[
\begin{align*}
\Gamma; P \parallel x : \Gamma & \quad (R\text{-NUM}) \\
\Gamma; P \parallel b : \Gamma & \quad (R\text{-BOOLEAN}) \\
\Gamma; P \parallel \beta : \Gamma & \quad (R\text{-VAR}) \\
\Gamma; P \parallel \eta : \Gamma & \quad (R\text{-RAND}) \\
P \neq \emptyset & \\
\Gamma; P \parallel \eta : \text{refine}(\Gamma, \eta, \alpha) & \quad (R\text{-REFINE}) \\
\Gamma; P \parallel \eta \parallel \gamma & \\
\Gamma; P \parallel \eta \parallel \chi & \\
\end{align*}
\]

Inference algorithm

Our inference algorithm refines the typing environment using the **refine** function. This function replaces all occurrences of $\alpha$ in $\Gamma$ to $d$ when $\Gamma(x) = \alpha$ is a variable to be inferred. 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 $x$.

Consider Figure 1 where all local variables are to be inferred. We first run the typing algorithm. The first refinement happens at line 2, where the distance of $\bar{T}$ is refined to $\alpha$, the distance variable of $\eta_1$. At line 3, $c_1$, $c_2$ and $i$ are refined to distance $0$. In the loop body, $\eta_2$ is refined to $\hat{q}[i] + \eta_2 \geq T'/\beta : \gamma$ at line 6. At line 8, **refine($\Gamma, c_1, 0$)** returns $\Gamma$ since $0 = 0$ is always true. Similar for the "else" branch and line 12. Hence, the environment after line 12 is a fixed point for the loop body. Hence, the typing environment after refinement

\[
\begin{align*}
\Gamma; P \parallel x : \Gamma & \quad (R\text{-VAR}) \\
\Gamma; P \parallel : \Gamma & \quad (R\text{-RAND}) \\
\Gamma; P \parallel e : \Gamma' & \quad (R\text{-CONS}) \\
\Gamma; P \parallel e : \Gamma' & \quad (R\text{-NEG}) \\
\Gamma; P \parallel e : \Gamma' & \quad (R\text{-EQ}) \\
\Gamma; P \parallel e : \Gamma' & \quad (R\text{-IF}) \\
\Gamma; P \parallel e : \Gamma' & \quad (R\text{-WHILE}) \\
\end{align*}
\]

Refinement rules for commands

\[
\begin{align*}
\Gamma \parallel \text{skip} & \quad (R\text{-SKIP}) \\
\Gamma \parallel \text{return} e : \Gamma & \quad (R\text{-RETURN}) \\
\end{align*}
\]

As we described informally above, Rule (R-ASGN-REF) refines the distance of $x$ using the **refine** function when its distance is not given. For expressions, the refinement algorithm propagates context information $P$ to subexpressions. Hence, each rule for expression has the form of $\Gamma, P \parallel e : \Gamma'$, where $P$ is a predicate that may appear in a dependent type. The context information is used to refine distance of a random variable $\eta$. Note that the refinement is not needed for a normal variable $x$ since the "shape" of $x$ is either provided or has been refined already when $x$ is initialized. However, this is not true for a random variable: $\eta$ can have any distance expression according to rule (T-ODOT).

The rule (R-WHILE) assumes that a fixed point exists. Based on the definition of the **refine** function, a fixed point can be efficiently computed as follows. We define $\leq$ as the lifted relation based on a point-wise lattice (for each variable) where: $\forall \alpha, \beta, \alpha \leq \beta \Leftrightarrow \alpha \leq \beta \leq \alpha$. We can compute a fixed point $\Gamma_0 = \Gamma^0 \vdash e : \Gamma^0, \Gamma^1 \vdash e : \Gamma^1, \Gamma^2 \vdash e : \Gamma^2, \ldots$ until $\Gamma^i \vdash e : \Gamma^i$ for some $i$. Based on the definition of the **refine** function, it is easy to check that $\Gamma^n \leq \Gamma^{n+1}$ and the computation terminates. The latter is true since after each refinement, either the number of distance variables is reduced by one, or there is at most one more distance ($*$) to be refined to.

**Example** We consider the 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 $\bar{T}$ is refined to $\alpha$, the distance variable of $\eta_1$. At line 3, $c_1$, $c_2$ and $i$ are refined to distance 0. In the loop body, $\eta_2$ is refined to $\hat{q}[i] + \eta_2 \geq T'/\beta : \gamma$ at line 6. At line 8, **refine($\Gamma, c_1, 0$)** returns $\Gamma$ since $0 = 0$ is always true. Similar for the "else" branch and line 12. Hence, the environment after line 12 is a fixed point for the loop body. Hence, the typing environment after refinement

\[
\begin{align*}
\Gamma \parallel \eta & \quad (R\text{-LAPLACE}) \\
\end{align*}
\]
Figure 10. The target program with unknown type variables. The
instrumented statements are un-commented.

\[ \Gamma(\text{count1}) = \Gamma(\text{count2}) = \Gamma(i) = 0, \Gamma(\hat{T}) = \Gamma(\eta_1) = \alpha \]

Type checking with distance variables With potential type variables
in the refined environment \( \Gamma \), the type system collects con-
straints during type checking, and tries to solve the collected con-
straints where the type variables are existentially qualified. For ex-
ample, using the refined typing environment above, type check-
ing the Sparse Vector method generates only one (nontrivial) con-
straint: \( \forall q[i], \eta_1, \eta_2, T \in \mathbb{R}. \quad (\neg 1 \leq \hat{q}_i \leq 1) \Rightarrow q[i] + \eta_2 \geq \hat{T} \quad \land \quad \eta_1 + \eta_2 \geq \hat{T}\beta \geq \gamma) \geq \hat{T} + \alpha \)

In fact, except the optimal solution we annotated in Figure 11
(i.e., \( \alpha = 1, \beta = 2, \gamma = 0 \)), 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 \( v_k = \frac{\alpha}{2} + \frac{\beta}{2} + c_2 \times \frac{\gamma}{2N} \) 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 \( c_2 \) is not bounded:

\[ \min \left( \frac{\alpha}{2} + \frac{\beta}{2} + M \times \gamma \right) \) such that

\[ (\forall q[i], \eta_1, \eta_2, T \in \mathbb{R}. \quad (\neg 1 \leq \hat{q}_i \leq 1) \Rightarrow q[i] + \eta_2 \geq \hat{T} \quad \land \quad \eta_1 + \eta_2 \geq \hat{T}\beta \geq \gamma) \geq \hat{T} + \alpha \)

Using a MaxSMT solver \( \mu Z \) [Bjørner and Phan 2014; Bjørner et al.
2015]. we successfully find the optimal solution for the type vari-
ables: \( \alpha = 1, \beta = 2, \gamma = 0 \). 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. However, we
leave systematic research in this direction as future work.

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 AutoPriv AutoPriv can easily verify this
numerical variant from the scratch, in a very similar way as verifying
the Sparse Vector method. However, here we focus on another
interesting scenario of using AutoPriv: the programmer (or algorithm
designer) has already verified the Sparse Vector method using
AutoPriv, and she is now exploring its variations. This is a common
scenario for algorithm designers. We show that since AutoPriv
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 \( \eta_3 \), and the loop

Figure 11. The Numerical Sparse Vector method.
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 \( \eta \), which is trivially solved to be \( \text{num}_\eta [i] \). Moreover, AutoPriv 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 \( v_i \leq \epsilon \). 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 AutoPriv.

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 inadvertently re-used the same noise used for the “if” test (Line 7) instead of drawing new noise when updating the noisy query answer. In other words, Lines 5–8 in Figure 1 are replaced with:

\[
\eta_2 := \text{Lap}\left(2N/e\right) ; \\
\hat{q} = q[i] + \eta_2 \\
\text{if } (q \geq \hat{T}) \text{ then} \\
\text{cons } (\hat{q}) \text{ out} ;
\]

For this incorrect variant, the definition of \( \hat{q} \) generates a constraint \( (d = q[i] + \alpha) \), where \( \Gamma(\hat{q}) = \text{num}_\eta \) and \( \Gamma(\eta_2) = \text{num}_\eta \). Moreover, \( (\text{cons } (\hat{q}) \text{ out}) \) generates a constraint \( (d = 0) \) by rule \( \text{(T-CONS)} \). Hence, it must be true that \( \Gamma(\hat{q}) = \text{num}_\eta \) and \( \Gamma(\eta_2) = \text{num}_\eta \).

In fact, the inference engine of AutoPriv is able to infer types of this incorrect variant. Moreover, \( \eta_2 := \text{Lap}\left(2N/e\right) \) is transformed to \( (\text{havoc } \eta_2; v_i := v_i + [q[i]](e/2N)) \). However, we cannot prove that the incorrect variant is \( \epsilon \)-private for any \( \epsilon \). The reason is that \( v_i \) in the transformed program is clearly not bounded by any constant \( \epsilon \), since \( v_i \) increments by \( e/2N \) in the worst case in each loop iteration, but the number of iterations is unbounded.

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.

6.2 Smart Sum

We next study a smart summation algorithm verified previously (with heavy annotations) in Barthe et al. (2013, 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: \( q[0], q[0] + q[1], \ldots, \sum_{i=0}^{T} q[i] \). A 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 \( \hat{q}[i] = q[i] + \text{Lap}(1/\epsilon) \) for each \( i \) and output \( q[0], q[0] + \hat{q}[1], \ldots, \sum_{i=0}^{T} \hat{q}[i] \). 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 \( G_1 = \{ q[0], q[1], \ldots, q[M - 1] \} \), \( G_2 = \{ q[M], q[M + 1], \ldots, q[2M - 1] \} \), etc. Then it maintains 2 levels of noisy counts: (1) the noisy bits \( \hat{q}[i] = q[i] + \text{Lap}(1/\epsilon) \) for each \( i \), and (2) the noisy block sums \( \hat{G}_j = \sum_{i \in G_j} \hat{q}[i] + \text{Lap}(1/\epsilon) \) 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 \( \ell + 1 = x M + c \) where \( x = \lfloor \ell + 1 / M \rfloor \) 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: \( \hat{G}_1 + \hat{G}_2 + \cdots + \hat{G}_x + \sum_{j=0}^{c} \hat{q}[x B + j] \). 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 \( \hat{G}_j \) rather than the sum of its noisy bits).

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 \( 2\epsilon \)-privacy where \( \epsilon \) is a function parameter.

Verification using AutoPriv

AutoPriv 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 \( 2\epsilon \)-private, it is sufficient to verify that the postcondition \( v_i \leq 2\epsilon \) 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

```
function SMARTSUM (\( \text{num}_\eta [i], M, T \) list \( \text{num}_\eta [q] \))
returns list \( \text{num}_\eta [q] \)
precondition \( \forall i, 1 \leq (\hat{q}[i]) \leq 1 \wedge (v_i, \hat{q}[i] \neq 0 \Rightarrow (\forall j \neq i, \hat{q}[j] = 0) \)
num_\eta [i] := n; i := 0; sum := 0; 
1 while i < T 
2 if (i + 1) mod M = 0 then 
3 \( \eta_2 := \text{Lap}(1/\epsilon) \) 
4 havoc \( \eta_1; v_i := v_i + [\hat{q}[i]](e/2N) \) 
5 n := n + sum + q[i] + \hat{eta}; 
6 next := n; 
7 sum := 0; 
8 cons next out; 
9 else 
10 \( \eta_2 := \text{Lap}(1/\epsilon) \) 
11 next := next + q[i] + \hat{eta}; 
12 sum := sum + q[i]; 
13 cons next out; 
14 i := i + 1;
```

Figure 12. The SmartSum algorithm.
by providing the following type for \( \eta \): \( \Gamma(\eta) = \eta \cdot d \) where
\[
d = (\eta \leq t)?(\hat{\ell} \geq 0?0: (\hat{\ell}/t))
\]
\[
: (\ell \leq 0?0: (\ell/t))
\]  

During type checking, rule \( (T-ODOT) \) checks the following constraint for the branch condition \( \eta \leq t \Rightarrow \eta \cdot d \leq t + \hat{\ell} \), 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 Lemma2 the privacy cost of this subtle example is provably bounded by the transformed cost formula in the transformed program[1].

### 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 that privately release final query answers. AutoPriv, on the other hand, verifies sophisticated privacy-preserving mechanisms that releases those final answers. HOARE 2 (Barthe et al. 2015) has the ability to relate a pair of expressions via relational assertions that appear as refinements in types. Hence, it can verify mechanisms that privately release final query answers. However, HOARE 2 incurs heavy annotation burden on programmers. Moreover, it 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. 2016) 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. (2016) can verify the Sparse Vector method. Compared with AutoPriv, 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 AutoPriv 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 AutoPriv 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 AutoPriv.

#### Other language-based methods for differential privacy
Several dynamic tools exist for enforcing differential privacy. PINQ (McSherry)

---

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 that privately release final query answers. AutoPriv, on the other hand, verifies sophisticated privacy-preserving mechanisms that releases those final answers. HOARE 2 (Barthe et al. 2015) has the ability to relate a pair of expressions via relational assertions that appear as refinements in types. Hence, it can verify mechanisms that privately release final query answers. However, HOARE 2 incurs heavy annotation burden on programmers. Moreover, it 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. 2016) 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. (2016) can verify the Sparse Vector method. Compared with AutoPriv, 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 AutoPriv 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 AutoPriv 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 AutoPriv.

**Other language-based methods for differential privacy**

Several dynamic tools exist for enforcing differential privacy. PINQ (McSherry)

---

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 that privately release final query answers. AutoPriv, on the other hand, verifies sophisticated privacy-preserving mechanisms that releases those final answers. HOARE 2 (Barthe et al. 2015) has the ability to relate a pair of expressions via relational assertions that appear as refinements in types. Hence, it can verify mechanisms that privately release final query answers. However, HOARE 2 incurs heavy annotation burden on programmers. Moreover, it 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. 2016) 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. (2016) can verify the Sparse Vector method. Compared with AutoPriv, 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 AutoPriv 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 AutoPriv 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 AutoPriv.

**Other language-based methods for differential privacy**

Several dynamic tools exist for enforcing differential privacy. PINQ (McSherry)

---

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 that privately release final query answers. AutoPriv, on the other hand, verifies sophisticated privacy-preserving mechanisms that releases those final answers. HOARE 2 (Barthe et al. 2015) has the ability to relate a pair of expressions via relational assertions that appear as refinements in types. Hence, it can verify mechanisms that privately release final query answers. However, HOARE 2 incurs heavy annotation burden on programmers. Moreover, it 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. 2016) 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. (2016) can verify the Sparse Vector method. Compared with AutoPriv, 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 AutoPriv 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 AutoPriv 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 AutoPriv.

**Other language-based methods for differential privacy**

Several dynamic tools exist for enforcing differential privacy. PINQ (McSherry)
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 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, AutoPriv currently only verifies \( \epsilon \)-privacy, which has a nice point-wise property. We leave extending AutoPriv to \((\epsilon, \delta)\)-privacy as future work.

References


A. Roth.

The sparse vector technique.


I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel.


M. C. Tschantz, D. Kaynar, and A. Datta.

Formal verification of differential privacy for interactive systems (extended abstract).


M. Wand.


L. Xu, K. Chatzikokolakis, and H. Lin.


D. Zhang and A. C. Myers.


J. Zhang, X. Xiao, and X. Xie.


### Appendix

#### A. Soundness proof

To prove this theorem, we first need a few auxiliary lemmas.

**Lemma 3.**

\[ \forall m_1, m_2, \Gamma \vdash m_1 \rightarrow e : B_d, \text{ we have} \]

\[ \forall m. \text{unit} m_1(m) = \text{unit} m_2(\Gamma(m)) \]

**Proof.** By the fact that \( \Gamma \) is a function. \( \square \)

**Lemma 4 (Expression).**

\[ \forall e, m_1, m_2, \Gamma \vdash m_1, m_2 \rightarrow e : B_d, \] \[ \text{we have} \]

\[ [e]_{m_1} + [d]_{m_1} = [e]_{m_2} \]

**Proof.** Induction on the structure of \( e \). Interesting cases are follows.

When \( e \) is \( x \) or \( \eta \), result is true by the definition of \( m_1, m_2 \).

When \( e \) is \( e_1 + e_2 \), \( e_1 \vdash e_1 : B_d^i \) for \( i \in \{1, 2\} \).

By induction hypothesis, we have \( [e_1]_{m_1} = [e_1]_{m_2} \) and \( \Gamma \vdash e_1 : B_d^i \) for \( i \in \{1, 2\} \). By rule (T-ODOT), for any memory \( m_1 \), \( [e_1 \circ e_2]_m = ([e_1 \circ e_2]_m) \).

Hence, \( [e_1 \circ e_2]_{m_1} = [e_1 \circ e_2]_{m_2} \).

By definition of \( e \) and \( \Gamma \), we have \( [e]_{m_1} + [d]_{m_1} = [e]_{m_2} + [d]_{m_2} \).

**Proof of Lemma 3**

\[ \forall e, m_1, m_2, m, \Gamma \vdash e \rightarrow e' \land m_1 \Gamma m_2, \text{ we have} \]

\[ [e]_{m_1}(m) \leq \exp(\max([e']_{m_1}(m)) [c]_{m_2}(\Gamma(m))) \]

**Proof.** By structural induction on \( e \).

- **Case skip:** \( e' = \text{skip by typing rule}. \) Hence, \( \max([e']_{m_1}(m)) = 0. \) Desired result is true by Lemma 3.

- **Case case:** \( e := e \) transformation, we have \( \max([e']_{m_1}(m)) = 0. \) Hence, by the semantics and Lemma 3 it is sufficient to show that \( \Gamma m_1 \Gamma m_2 \) where \( m_1 = m_1 \{[e]_{m_1}/x\} \) and \( m_2 = m_2 \{[e]_{m_2}/x\} \).

We first show \( m_1(x) + [d]_{m_1} = m_2(x) \), where \( \Gamma \vdash x : c \).

Consider \( \Gamma(x) = B_d \). By typing rule, we have \( \Gamma \vdash e : B_d \).

By Lemma 3 \( [c]_{m_1} + [d]_{m_1} = [e]_{m_2} m_2 \).

Hence, \( m_1(x) + [d]_{m_1} = m_2(x) \).

Since \( \Gamma \) may only depend on immutable variables in this case, \( [d]_{m_1} = [d]_{m_2} \).

So \( m_1(x) + [d]_{m_2} = m_2(x) \) as desired. Consider \( \Gamma(x) = B_d \). In this case, \( \Gamma \vdash x : B_d \). \( m_1(x) + [d]_{m_2} = m_2(x) \) by semantics.

Second, we show \( m_1(y) + [d]_{m_1} = m_2(y) \), where \( \Gamma \vdash y : c \) for \( y \in \text{dom}(\Gamma) \land y \neq x \).

When \( y \in \text{Var} \), its type cannot depend on \( x \), which is mutable. So the desired result is true. For \( \eta \in H \), its type only depends on the memory state when \( \eta \) is used. So the desired result is true as well.

- **Case if e then c1 else c2:** by typing rule, \( \Gamma \vdash e : \text{bool} \).

By Lemma 3 \( [e]_{m_1} = [e]_{m_2} m_2 \).

Hence, the same branch is taken in \( m_1 \) and \( m_2 \). Desired result is true by induction hypothesis.

- **Case c1; c2:** For any \( m \) such that \( [c1; c2]_{m_1}(m) \neq 0 \), there exists some \( m_1 \) such that \( [c1]_{m_1}(m_1) \neq 0 \wedge [c2]_{m_1}(m_1) \neq 0 \)

By hypothesis,

\[ [c1]_{m_1}(m_1) \leq \exp(e_1) [c1]_{\Gamma(m_1)}(\Gamma(m_1)) \]

\[ [c2]_{m_1}(m_1) \leq \exp(e_2) [c2]_{\Gamma(m_1)}(\Gamma(m_1)) \]

where \( e_1 = \max([c_1]_{m_1}(m)) \) and \( e_2 = \max([c_2]_{m_1}(m)) \).

Hence,

\[ [c1]_{m_1}(m_1) \cdot [c2]_{m_1}(m_1) \leq \exp(e_1 + e_2) [c1]_{\Gamma(m_1)}(\Gamma(m_1)) \cdot [c2]_{\Gamma(m_1)}(\Gamma(m_1)) \]

Notice that \( m_1 \leftrightarrow (e_1) \not\in [c1]_{m_1}(m_1) \) and \( m_2 \leftrightarrow (e_2) \not\in [c2]_{m_1}(m_1) \) since \( e_1 \) and \( e_2 \) maximize privacy costs among consistent executions by definition. Hence, \( m \leftrightarrow (e_1 + e_2) \not\in [c1; c2]_{m_1}(m_1) \).

Therefore, \( e_1 + e_2 \leq \max(c1; c2)_{m_1}(m_1) \).

So for any \( m \),

\[ [c1; c2]_{m_1}(m_1) = \sum_{m_1} [c1]_{m_1}(m') \cdot [c2]_{m_1}(m') \]

\[ \leq \exp(e_0) \sum_{m_1} [c1]_{\Gamma(m_1)}(\Gamma(m_1)) \cdot [c2]_{\Gamma(m_1)}(\Gamma(m_1)) \]

\[ \leq \exp(e_0) \sum_{m_1} [c1]_{\Gamma(m_1)}(\Gamma(m_1)) \cdot [c2]_{\Gamma(m_1)}(\Gamma(m_1)) \]

\[ \leq \exp(e_0) \sum_{m_1} [c1]_{\Gamma(m_1)}(\Gamma(m_1)) \]

where \( e_0 = \max(c1; c2)_{m_1}(m_1) \). Notice that the change of variable in the second to last inequality only holds when \( \Gamma(x) \) is an injective (but not necessarily onto) mapping.

- **Case while e do c:** let \( W = \text{while e do c} \). By typing rule, \( \Gamma \vdash e : \text{bool} \).

By Lemma 3 \( [e]_{m_1} = [e]_{m_2} m_2 \) for any \( m_1 \) \( m_2 \).

We proceed by natural induction on the number of loop iterations (denoted by \( i \)) under \( m_1 \).

When \( i = 0 \), \( [b]_{m_1} = \text{false} \). So \( [b]_{m_2} = \text{false} \) since \( m_1 \Gamma m_2 \).

By semantics, \( [W]_{m_1} = \text{unit} m_1 \) and \( [W]_{m_2} = \text{unit} m_2 \).

and the latter \( W \) iterates for \( i \) times. By induction hypothesis and a similar argument of the sequential case, \( [W]_{m_1}(m_1) \leq \exp(\max(W)_{m_1}) [W]_{m_1}(\Gamma(m_1)) \).

- **Case let \( \eta := \text{Lap} \) of \( m \):** \( \mu_e \) is the Laplace distribution with a scale factor of \( r \), we have

\[ \forall v, d \in R, \mu_e(v) \leq \exp(d \times r) \mu_e(v + d) \]

When $\exists v. m = m_1\{v/\eta\}$, $[\eta := \text{Lap } r]_m (m) = 0$ by semantics. Hence, desired inequality is trivial. When $m = m_1\{v/\eta\}$ for some $v$, we have

\[
\begin{align*}
[\eta := \text{Lap } r]_m (m) &= \mu_e(v) \\
&\leq \exp(|d| \cdot r)\mu_e(v + d)
\end{align*}
\]

Since $m_1 \Gamma m_2$, for any $v, m_1\{v/\eta\} \Gamma m_2\{v + d/\eta\}$. That is, $\Gamma(m) = m_2\{v + d/\eta\}$. By semantics, $[\eta := \text{Lap } r]_m (\Gamma(m)) = [\eta := \text{Lap } r]_{m_2} (m_2\{v + d/\eta\}) = \mu_e(v + d)$. Hence,

\[
[\eta := \text{Lap } r]_{m_1} (m) \leq \exp(|d| \cdot r) [\eta := \text{Lap } r]_{m_2} (\Gamma(m))
\]

By transformation, $\Gamma \vdash \eta := \text{Lap } r \rightarrow \text{havoc } \eta; v_e := v_e + |d| \cdot r$, where $\Gamma(\eta) = \epsilon$. Hence, $\max(c_1^{m_1\{v/\eta\}}) = |[d| \cdot r]_{m_1\{v/\eta\}} = |(d| \cdot r| \epsilon\{m_1\{v/\eta\}\}$. Therefore, we showed that

\[
[\eta := \text{Lap } r]_{m_1} (m) \leq \exp(\max(c_1^{m_1\{v/\eta\}}))[\eta := \text{Lap } r]_{m_2} (\Gamma(m))
\]

Proof of Theorem 2

\[
\forall \Gamma, c, c', x, e. \quad \Gamma \vdash (c; \text{return } x) \rightarrow (c'; \text{return } x) \text{ then } \Gamma(x) = B_0 \land c' \lll \Rightarrow c \text{ is } \epsilon\text{-private}
\]

Proof. By the soundness theorem (Theorem 1), we have for any $m_1 \Gamma m_2, \forall S \subseteq M, [c]_{m_1\{S\}} \leq \exp(e)[c]_{m_2} (\Gamma(S))$. Let $P = (c; \text{return } x)$. By semantics, for any value set $V \subseteq B,

\[
[P]_{m_1} (V) = [c]_{m_1 \{m \mid m(x) \in V\}} (1) \\
\leq \exp(e)[c]_{m_2 \{\Gamma(m) \mid \Gamma(m)(x) \in V\}} (2) \\
= \exp(e)[c]_{m_2 \{m \mid m(x) \in V\}} (3) \\
= \exp(e)[P]_{m_2} (V) (4)
\]

Where inequality (2) is true due to Theorem 1 and equality (3) is true since by Lemma 4 $\forall m \in M, v \in B, m(x) = v \Leftrightarrow \Gamma(m)(x) = v$. Hence,

\[
\{\Gamma(m) \mid \Gamma(m)(x) \in V\} = \{m \in M \mid m(x) \in V\}
\]

Hence, by definition of differential privacy, $c$ is $\epsilon$-private.

B. Formal semantics for the target language

The denotational semantics interprets a command $c$ in the target language (Figure 7) as a function $[c] : M \rightarrow P(M)$. The semantics of commands are formalized as follows.

\[
\begin{align*}
[\text{skip}]_m &= \{m\} \\
[c := e]_m &= \{m([e]_m/x)\} \\
[\text{havoc } x]_m &= \cup e \in E \{m(r/x)\} \\
[c_1; c_2]_m &= \cup m' \in [c_1]_m [c_2]_{m'} \\
[\text{if } e \text{ then } c_1 \text{ else } c_2]_m &= \begin{cases} [c_1]_m & \text{if } [e]_m = \text{true} \\
[c_2]_m & \text{if } [e]_m = \text{false} \\
\end{cases} \\
\text{while } e \text{ do } c \}_m &= w^* m \\
\text{where } w^* &= \{\text{fix} \lambda f. \lambda m. \text{if } [e]_m = \text{true} \\
&\quad \text{then } (\cup m' \in [c]_m f m') \text{ else } \{m\} \}
\end{align*}
\]

Accordingly, the Hoare logic rules for the target language is mostly standard:

\[
\begin{align*}
\{\Psi\} \text{skip }\{\Phi\} &\quad \{\Phi\} \text{skip }\{\Psi\} &\quad (\text{H-Skip}) \\
\{\forall x. \Phi \text{havoc } (x)\} \{\Psi\} &\quad \{\forall x. \Phi\} \{\Psi\} &\quad (\text{HAVOC}) \\
\{\Psi\} c_1 ; c_2 \{\Phi\} &\quad \{\Psi\} c_1 ; c_2 \{\Phi\} &\quad (\text{H-SEQ}) \\
\{\Psi\} \text{if } e \text{ then } c_1 \text{ else } c_2 \{\Phi\} &\quad \{\Psi\} \text{if } e \text{ then } c_1 \text{ else } c_2 \{\Phi\} &\quad (\text{H-IF}) \\
\{\Psi\} \text{while } e \text{ do } c \{\Phi\} &\quad \{\Psi\} \text{while } e \text{ do } c \{\Phi\} &\quad (\text{H-WHILE}) \\
\end{align*}
\]

C. Uniform distribution

Lemma 5 (UniformDist). The following typing rule is sound w.r.t.

\[
\Gamma(\eta) = \text{num}_{a,d} \\
\Gamma \vdash \eta := \text{Uniform}_{a,1} \rightarrow \text{havoc}_{a,1}\eta; v_e = v_e - \log(\epsilon + 1)
\]

Proof. When $\exists v. m = m_1\{v/\eta\}$ or $m(\eta) < -1$ or $m(\eta) > 1$, $\Gamma(\eta) = \text{Uniform}_{a,1} \{m\} = 0$ by semantics. Hence, desired inequality is trivial.

When $m = m_1\{v/\eta\}$ for some $-1 \leq v \leq 1$. Let $\mu_v = \text{Uniform}_{a,1} = [d]_{m_1}$, we have

\[
\begin{align*}
[\eta := \text{Uniform}_{a,1}]_{m_1}(m) &= \mu_v(v) \\
&= \int_{0}^{1} 1 \{x \leq v\} dx \\
&= \int_{0}^{1} 1 \{(d+1)x \leq (d+1)v\} dx \\
&= \int_{0}^{1} 1 \{(d+1)x \leq (d+1)v\} dx \\
&= \frac{1 + d}{d+1} \int_{0}^{1} 1 \{x \leq (d+1)v\} dx \\
&\leq \frac{1 + d}{1 + d} \int_{0}^{1} 1 \{x \leq (d+1)v\} dx \\
&\leq \exp(-\log(d+1))\mu_v(v + v \cdot d)
\end{align*}
\]

Since $m_1 \Gamma m_2$, for any $v, m_1\{v/\eta\} \Gamma m_2\{v + v \cdot d/\eta\}$. Hence

\[
\begin{align*}
[\eta := \text{Uniform}_{a,1}]_{m_1}(m) &= \exp(-\log(d+1))
\end{align*}
\]

By transformation, $\text{Uniform}_{a,1} \rightarrow \text{havoc}_{a,1}\eta; v_e = v_e - \log(\epsilon + 1)$, where $\Gamma(\eta) = \eta \cdot \epsilon$. Hence, $\max(c_1^{m_1\{v/\eta\}}) = \exp(-\log(d+1))\mu_v(v + v \cdot d)$. Therefore

\[
\begin{align*}
[\eta := \text{Uniform}_{a,1}]_{m_1}(m) &= \exp(-\log(d+1))\mu_v(v + v \cdot d)
\end{align*}
\]

\[
\Box
\]