Proving Differential Privacy with Shadow Execution

Yuxin Wang
Pennsylvania State University
University Park, PA, USA
yxwang@psu.edu

Zeyu Ding
Pennsylvania State University
University Park, PA, USA
zyding@psu.edu

Guanhong Wang
Pennsylvania State University
University Park, PA, USA
gpw5092@psu.edu

Daniel Kifer
Pennsylvania State University
University Park, PA, USA
dkifer@cse.psu.edu

Danfeng Zhang
Pennsylvania State University
University Park, PA, USA
zhang@cse.psu.edu

Abstract

Recent work on formal verification of differential privacy shows a trend toward usability and expressiveness – generating a correctness proof of sophisticated algorithm while minimizing the annotation burden on programmers. Sometimes, combining those two requires substantial changes to program logics: one recent paper is able to verify Report Noisy Max automatically, but it involves a complex verification system using customized program logics and verifiers.

In this paper, we propose a new proof technique, called shadow execution, and embed it into a language called ShadowDP. ShadowDP uses shadow execution to generate proofs of differential privacy with very few programmer annotations and without relying on customized logics and verifiers. In addition to verifying Report Noisy Max, we show that it can verify a new variant of Sparse Vector that reports the gap between some noisy query answers and the noisy threshold. Moreover, ShadowDP reduces the complexity of verification: for all of the algorithms we have evaluated, type checking and verification in total takes at most 3 seconds, while prior work takes minutes on the same algorithms.

CCS Concepts • **Software and its engineering → Formal software verification**.

Keywords Differential privacy; dependent types

ACM Reference Format:
Yuxin Wang, Zeyu Ding, Guanhong Wang, Daniel Kifer, and Danfeng Zhang. 2019. Proving Differential Privacy with Shadow Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314619

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314619

1 Introduction

Differential privacy is increasingly being used in industry [22, 27, 37] and government agencies [1] to provide statistical information about groups of people without violating their privacy. Due to the prevalence of errors in published algorithms and code [29], formal verification of differential privacy is critical to its success.

The initial line of work on formal verification for differential privacy (e.g., [6–10]) was concerned with increasing expressiveness. A parallel line of work (e.g., [31, 33, 35, 43]) focuses more on usability – on developing platforms that keep track of the privacy cost of an algorithm while limiting the types of algorithms that users can produce.

A recent line of work (most notably LightDP [42] and Synthesizing Coupling Proofs [2]) has sought to combine expressiveness and usability by providing verification tools that infer most (if not all) of the proof of privacy. The benchmark algorithms for this task were Sparse Vector [20, 29] and Report Noisy Max [20]. LightDP [42] was the first system that could verify Sparse Vector with very few annotations, but it could not verify tight privacy bounds on Report Noisy Max [20]. It is believed that proofs using *randomness alignment*, the proof technique that underpins LightDP, are often simpler, while *approximate coupling*, the proof technique that underpins [6–10], seems to be more expressive [2]. Subsequently, Albarghouthi and Hsu [2] produced the first fully automated system that verifies both Sparse Vector and Report Noisy Max. Although this new system takes inspiration from randomness alignment to simplify approximate coupling proofs, its verification system still involves challenging features such as first-order Horn clauses and probabilistic constraints; it takes minutes to verify simple algorithms. The complex verification system also prevents it from reusing off-the-shelf verification tools.

In this paper, we present ShadowDP, a language for verifying differentially private algorithms. It is based on a new proof technique called "shadow execution", which enables language-based proofs based on standard program logics.

Built on randomness alignment, it transforms a probabilistic program into a program in which the privacy cost is explicit; so that the target program can be readily verified by off-the-shelf verification tools. However, unlike LightDP, it can verify more challenging algorithms such as Report Noisy Max and a novel variant of Sparse Vector called Difference Sparse Vector. We show that with minimum annotations, challenging algorithms that took minutes to verify by [2] (excluding proof synthesis time) now can be verified within 3 seconds with an off-the-shelf model checker.

One extra benefit of this approach based on randomness alignment is that the transformed program can also be analyzed by standard symbolic executors. This appears to be an important property in light of recent work on detecting counterexamples for buggy programs [12, 17, 23, 24]. Producing a transformed program that can be used for verification of correct programs and bug-finding for incorrect programs is one aspect that is of independent interest (however, we leave this application of transformed programs to future work).

In summary, this paper makes the following contributions:

1. Shadow execution, a new proof technique for differential privacy (Section 2.4).
2. ShadowDP, a new imperative language (Section 3) with a flow-sensitive type system (Section 4) for verifying sophisticated privacy-preserving algorithms.
3. A formal proof that the verification of the transformed program by ShadowDP implies that the source code is ϵ-differentially private (Section 5).
4. Case studies on sophisticated algorithms showing that verifying privacy-preserving algorithms using ShadowDP requires little programmer annotation burden and verification time (Section 6).
5. Verification of a variant of Sparse Vector Technique that releases the difference between noisy query answers and a noisy threshold at the same privacy level as the original algorithm [20, 29]. To the best of our knowledge, this variant has not been studied before.

2 Preliminaries and Illustrating Example

2.1 Differential Privacy

Differential privacy is now considered a gold standard in privacy protections after recent high profile adoptions [1, 22, 27, 37]. There are currently several popular variants of differential privacy [13, 18, 19, 32]. In this paper, we focus on the verification of algorithms that satisfy pure differential privacy [19], which has several key advantages – it is the strongest one among them, the most popular one, and the easiest to explain to non-technical end-users [34].

Differential privacy requires an algorithm to inject carefully calibrated random noise during its computation. The purpose of the noise is to hide the effect of any person's record on the output of the algorithm. In order to present the formal definition, we first define the set of *sub-distributions* over a discrete 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 $\mu(E) \triangleq \sum_{e \in E} \mu(e)$. [1]

Differential privacy relies on the notion of adjacent databases (e.g., pairs of databases that differ on one record). Since differentially-private algorithms sometimes operate on query results from databases, we abstract adjacent databases as an adjacency relation $\Psi \subseteq A \times A$ on input query answers. For differential privacy, the most commonly used relations are: (1) each query answer may differ by at most n (for some number n), and (2) at most one query answer may differ, and that query answer differs by at most n. This is also known as *sensitivity* of the queries.

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

$$P(M(a_1) \in E) \leq e^{\epsilon} P(M(a_2) \in E).$$

2.2 Randomness Alignment

Randomness Alignment [42] is a simple yet powerful technique to prove differential privacy. Here, we illustrate the key idea with a fundamental mechanism for satisfying differential privacy–the *Laplace Mechanism* [30].

Following the notations in Section 2.1, we consider an arbitrary pair of query answers a_1 and a_2 that differ by at most 1, i.e., $-1 \leq a_1 - a_2 = c \leq 1$. The Laplace Mechanism (denoted as M) simply releases $a + \eta$, where η is a random noise sampled from the Laplace distribution of mean 0 and scale $1/\epsilon$; we use $p_{1/\epsilon}$ to denote its density function. The goal of randomness alignment is to "align" the random noise in two executions $M(a_1)$ and $M(a_2)$, such that $M(a_1) = M(a_2)$, with a corresponding privacy cost. To do so, we create an *injective* function $f : \mathbb{R} \to \mathbb{R}$ that maps η to $\eta + c$. Obviously, f is an alignment since $a_1 + \eta = a_2 + f(\eta)$ for any a_1, a_2. Then for an arbitrary set of outputs $E \subseteq \mathbb{R}$, we have:

$$\begin{aligned} P(M(a_1) \in E) &= \sum_{\eta | a_1 + \eta \in E} p_{1/\epsilon}(\eta) \leq \sum_{\eta | a_2 + f(\eta) \in E} p_{1/\epsilon}(\eta) \\ &\leq e^{\epsilon} \sum_{\eta | a_2 + f(\eta) \in E} p_{1/\epsilon}(f(\eta)) \\ &= e^{\epsilon} \sum_{\eta | a_2 + \eta \in E} p_{1/\epsilon}(\eta) = e^{\epsilon} P(M(a_2) \in E) \end{aligned}$$

The first inequality is by the definition of f: $a_1 + \eta \in E \implies a_2 + f(\eta) \in E$. The e^{ϵ} factor results from the fact that $p_{1/\epsilon}(\eta + c) / p_{1/\epsilon}(\eta) \leq e^{|c| \cdot \epsilon} \leq e^{\epsilon}$, when the Laplace distribution has scale $1/\epsilon$. The second to last equality is by change of variable from $f(\eta)$ to η in the summation, using the injectivity of f.

In general, let $H \in \mathbb{R}^n$ be the random noise vector that a mechanism M uses. A randomness alignment for $a_1 \Psi a_2$ is a function $f : \mathbb{R}^n \to \mathbb{R}^n$ such that:

[1] As is standard in this line of work (e.g., [8, 42]), we assume a sub-distribution instead of a distribution, since sub-distribution gives rise to an elegant program semantics in face of non-terminating programs [28].

1. $M(a_2)$ with noise $f(H)$ outputs the same result as $M(a_1)$ with noise H (hence the name Randomness Alignment).
2. f is injective (this is to allow change of variables).

2.3 The Report Noisy Max Algorithm

To illustrate the challenges in proving differential privacy, we consider the Report Noisy Max algorithm [20], whose source code is shown on the top of Figure 1. It can be used as a building block in algorithms that iteratively generate differentially private synthetic data by finding (with high probability) the identity of the query for which the synthetic data currently has the largest error [25].

The algorithm takes a list q of query answers, each of which differs by at most 1 if the underlying database is replaced with an adjacent one. The goal is to return the index of the largest query answer (as accurately as possible subject to privacy constraints).

To achieve differential privacy, the algorithm adds appropriate Laplace noise to each query. Here, `Lap` $(2/\epsilon)$ draws one sample from the Laplace distribution with mean zero and a scale factor $(2/\epsilon)$. For privacy, the algorithm uses the noisy query answer $(q[i] + \eta)$ rather than the true query answer $(q[i])$ to compute and return the *index* of the maximum (noisy) query answer. Note that the return value is listed right below the function signature in the source code.

Informal proof using randomness alignment Proofs of correctness of Report Noisy Max can be found in [20]. We will start with an informal correctness argument, based on the *randomness alignment* technique (Section 2.2), to illustrate subtleties involved in the proof.

Consider the following two databases D_1, D_2 that differ on one record, and their corresponding query answers:

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

Suppose in one execution on D_1, the noise added to $q[0], q[1], q[2]$ is $\alpha_0^{(1)} = 1$, $\alpha_1^{(1)} = 2$, $\alpha_2^{(1)} = 1$, respectively. In this case, the noisy query answers are $q[0] + \alpha_0^{(1)} = 2$, $q[1] + \alpha_1^{(1)} = 4$, $q[2] + \alpha_2^{(1)} = 3$ and so the algorithm returns 1, which is the index of the maximum noise query answer of 4.

Aligning randomness As shown in Section 2.2, we need to create an injective function of random bits in D_1 to random bits in D_2 to make the output the same. Recall that $\alpha_0^{(1)}, \alpha_1^{(1)}, \alpha_2^{(1)}$ are the noise added to D_1, now let $\alpha_0^{(2)}, \alpha_1^{(2)}, \alpha_2^{(2)}$ be the noise added to the queries $q[0], q[1], q[2]$ in D_2, respectively. Consider the following injective function: for any query except for $q[1]$, use the same noise as on D_1; add 2 to the noise used for $q[1]$ (i.e., $\alpha_1^{(2)} = \alpha_1^{(1)} + 2$).

In our running example, execution on D_2 with this alignment function would result in noisy query answers $q[0] + \alpha_0^{(2)} = 3$, $q[1] + \alpha_1^{(2)} = 5$, $q[2] + \alpha_2^{(2)} = 3$. Hence, the output once again is 1, the index of query answer 5.

function NoisyMax (ϵ, size : num$_{\langle 0,0\rangle}$; q : list num$_{\langle *,*\rangle}$)
returns max : num$_{\langle 0,*\rangle}$
precondition $\forall i \geq 0.\ -1 \leq \widehat{q}^{\circ}[i] \leq 1 \ \wedge\ \widehat{q}^{\dagger}[i] = \widehat{q}^{\circ}[i]$

```
1  i := 0; bq := 0; max := 0;
2  while (i < size)
3      η := Lap (2/ε), Ω ? † : ∘, Ω ? 2 : 0 ;
4      if (q[i] + η > bq ∨ i = 0)
5        max := i;
6        bq := q[i] + η;
7      i := i + 1;
```

The transformed program (slightly simplified for readability), where underlined commands are added by the type system:

```
1  v_ε := 0;
2  i := 0; bq := 0; max := 0;
3  bq̃° := 0; bq̃† := 0;
4  while (i < size)
5      assert (i < size);
6      havoc η; v_ε := Ω ? (0 + ε) : (v_ε + 0);
7      if (q[i] + η > bq ∨ i = 0)
8        assert (q[i] + q̂°[i] + η + 2 > bq + bq̃† ∨ i = 0);
9        max := i;
10       bq̃† := bq + bq̃† - (q[i] + η);
11       bq := q[i] + η;
12       bq̃° := q̂°[i] + 2;
13     else
14       assert (¬(q[i] + q̂°[i] + η + 0 > bq + bq̃° ∨ i = 0));
15     // shadow execution
16     if (q[i] + q̂†[i] + η > bq + bq̃† ∨ i = 0)
17       bq̃† := q[i] + q̂†[i] + η − bq;
18     i := i + 1;
```

Figure 1. Verifying Report Noisy Max with ShadowDP. Here, q is a list of query answers from a database, and max is the query index of the maximum query with Laplace noise generated at line 3. To verify the algorithm on the top, a programmer provides function specification as well as annotation for sampling command (annotations are shown in gray, where Ω represents the branch condition). ShadowDP checks the source code and generates the transformed code (at the bottom), which can be verified with off-the-shelf verifiers.

In fact, we can prove that under this alignment, *every execution on D_1 where 1 is returned* would result in an execution on D_2 that produces the same answer due to two facts:

1. On D_1, $q[1] + \alpha_1^{(1)}$ has the maximum value.
2. On D_2, $q[1] + \alpha_1^{(2)}$ is greater than $q[1] + \alpha_1^{(1)} + 1$ on D_1 due to $\alpha_1^{(2)} = \alpha_1^{(1)} + 2$ and the adjacency assumption.

Hence, $q[1] + \alpha_1^{(2)}$ on D_2 is greater than $q[i] + \alpha_i^{(1)} + 1$ on D_1 for any i. By the adjacency assumption, that is the same as $q[1] + \alpha_1^{(2)}$ is greater than any $q[i] + \alpha_i^{(2)}$ on D_2. Hence,

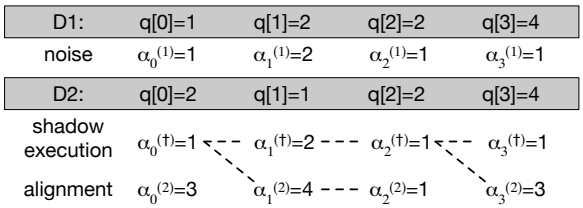

Figure 2. Selective alignment for Report Noisy Max

based on the same argument in Section 2.2, we can prove that the Report Noisy Max algorithm is ϵ-private.

Challenges Unfortunately, the alignment function above only applies to executions on D_1 where index 1 is returned. If there is one more query $q[3] = 4$ and the execution gets noise $\alpha_3^{(1)} = 1$ for that query, the execution on D_1 will return index 3 instead of 1. To align randomness on D_2, we need to construct a different alignment function (following the construction above) that adds noise in the following way: for any query except for $q[3]$, use the same noise as on D_1; add 2 to the noise used for $q[3]$ (i.e., $\alpha_3^{(2)} = \alpha_3^{(1)} + 2$). In other words, to carry out the proof, the alignment for each query depends on the queries and noise yet to happen *in the future*.

One approach of tackling this challenge, followed by existing language-based proofs of Report Noisy Max [2, 8], is to use the pointwise lifting argument: informally, if we can show that for any value i, execution on D_1 returns value i implies execution on D_2 returns value i (with a privacy cost bounded by ϵ), then a program is ϵ-differential private. However, this argument does not apply to the randomness alignment technique. Moreover, doing so requires a customized program logic for proving differential privacy.

2.4 Approach Overview

In this paper, we propose a new proof technique "shadow execution", which enables language-based proofs based on *standard* program logics. The key insight is to track a *shadow execution* on D_2 where the *same noise* is always used as on D_1. For our running example, we illustrate the shadow execution in Figure 2, with random noise $\alpha_0^{(\dagger)}$, $\alpha_1^{(\dagger)}$ and so on. Note that the shadow execution uses $\alpha_i^{(\dagger)} = \alpha_i^{(1)}$ for all i.

With the shadow execution, we can construct a randomness alignment for each query i as follows:

Case 1: Whenever $q[i] + \alpha_i^{(1)}$ is the maximum value so far on D_1 (i.e., *max* is updated), we use the alignments on *shadow execution* for all previous queries but a noise $\alpha_i^{(1)} + 2$ for $q[i]$ on D_2.

Case 2: Whenever $q[i] + \alpha_i^{(1)}$ is smaller than or equal to any previous noise query answer (i.e., *max* is not updated), we keep the previous alignments for previous queries and use noise $\alpha_i^{(1)}$ for $q[i]$ on D_2.

We illustrate this construction in Figure 2. After seeing $q[1]$ on D_1 (Case 1), the construction uses noise in the shadow execution for previous query answers, and uses $\alpha_1^{(1)} + 2 = 4$ as the noise for $q[1]$ on D_2. After seeing $q[2]$ on D_1 (Case 2), the construction reuses alignments constructed previously, and use $\alpha_2^{(1)} = 1$ as the noise for $q[2]$. When $q[3]$ comes, the previous alignment is abandoned; the shadow execution is used for $q[0]$ to $q[2]$. It is easy to check that this construction is correct for any subset of query answers seen so far, since the resulting alignment is exactly the same as the informal proof above, when the index of maximum value is known.

Randomness alignment with shadow execution To incorporate the informal argument above to a programming language, we propose ShadowDP. We illustrate the key components of ShadowDP in this section, as shown in Figure 1, and detail all components in the rest of this paper.

Similar to LightDP [42], ShadowDP embeds randomness alignments into types. In particular, each *numerical variable* has a type in the form of $\mathsf{num}_{\langle \mathbb{n}^\circ, \mathbb{n}^\dagger \rangle}$, where $\mathbb{n}^\circ$ and $\mathbb{n}^\dagger$ represent the "difference" of its value in the aligned and shadow execution respectively. In Figure 1, non-private variables, such as $\epsilon, size$, are annotated with distance 0. For private variables, the difference could be a constant or an expression. For example, the type of q along with the precondition specifies the adjacency relation: each query answer's difference is specified by $*$, which is desugared to a special variable $\widehat{q}^\circ[i]$ (details discussed in Section 4). The precondition in Figure 1 specifies that the difference of each query answer is bounded by 1 (i.e., query answers have sensitivity of 1).

ShadowDP reasons about the aligned and shadow executions in isolation, with the exception of sampling commands. A sampling command (e.g., line 3 in Figure 1) constructs the aligned execution by either using values from the aligned execution so far (symbol $\circ$), or switching to values from the shadow execution (symbol $\dagger$). The construction may depend on program state: in Figure 1, we switch to shadow values iff $q[i] + \eta$ is the max on D_1. A sampling command also specifies the alignment for the generated random noise.

With function specification and annotations for sampling commands, the type system of ShadowDP automatically checks the source code. If successful, it generates a non-probabilistic program (as shown at the bottom of Figure 1) with a distinguished variable $\mathbf{v}_\epsilon$. The soundness of the type system ensures the following property: if $\mathbf{v}_\epsilon$ is bounded by some constant ϵ in the transformed program, then the original program being verified is ϵ-private.

Benefits Compared with previous language-based proofs of Report Noisy Max [2, 8] (both are based on the pointwise lifting argument), ShadowDP enjoys a unique benefit: the transformed code can be verified based on *standard* program semantics. Hence, the transformed (non-probabilistic) program can be further analyzed by existing program verifiers and other tools. For example, the transformed program in Figure 1 is verified with an off-the-shelf tool CPAChecker[11]

Reals	r	$\in$	$\mathbb{R}$
Normal Vars	x	$\in$	*NVars*
Random Vars	η	$\in$	*RVars*
Linear Ops	$\oplus$	::=	$+ \mid -$
Other Ops	$\otimes$	::=	$\times \mid /$
Comparators	$\odot$	::=	$< \mid > \mid = \mid \leq \mid \geq$
Bool Exprs	$\mathbb{b}$	::=	$\texttt{true} \mid \texttt{false} \mid x \mid \neg \mathbb{b} \mid \mathbb{n}_1 \odot \mathbb{n}_2$
Num Exprs	$\mathbb{n}$	::=	$r \mid x \mid \eta \mid \mathbb{n}_1 \oplus \mathbb{n}_2 \mid$ $\mathbb{n}_1 \otimes \mathbb{n}_2 \mid \mathbb{b}?\ \mathbb{n}_1 : \mathbb{n}_2$
Expressions	e	::=	$\mathbb{n} \mid \mathbb{b} \mid e_1 :: e_2 \mid e_1[e_2]$
Commands	c	::=	$\texttt{skip} \mid x := e \mid \eta := g \mid c_1; c_2 \mid$ $\texttt{return}\ e \mid \texttt{while}\ e\ \texttt{do}\ (c) \mid$ $\texttt{if}\ e\ \texttt{then}\ (c_1)\ \texttt{else}\ (c_2)$
Distances	$\mathbb{d}$	::=	$\mathbb{n} \mid *$
Types	τ	::=	$\texttt{num}_{\langle \mathbb{d}^\circ, \mathbb{d}^\dagger \rangle} \mid \texttt{bool} \mid \texttt{list}\ \tau$
Var Versions	k	$\in$	$\{\circ, \dagger\}$
Selectors	$\mathcal{S}$	::=	$e\ ?\ \mathcal{S}_1 : \mathcal{S}_2 \mid k$
Rand Exps	g	::=	$\texttt{Lap}\ r\ ,\ \mathcal{S}\ ,\ \mathbb{n}_\eta$

Figure 3. ShadowDP: language syntax.

without any extra annotation within seconds. Although not explored in this paper, the transformed program can also be analyzed by symbolic executors to identify counterexamples when the original program is incorrect. We note that doing so will be more challenging in a customized logic.

3 ShadowDP: Syntax and Semantics

In this section, we present the syntax and semantics of ShadowDP, a simple imperative language for designing and verifying differentially private algorithms.

3.1 Syntax

The language syntax is given in Figure 3. Most parts of ShadowDP is standard; we introduce a few interesting features.

Non-probabilistic variables and expressions ShadowDP supports real numbers, booleans as well as standard operations on them. We use $\mathbb{n}$ and $\mathbb{b}$ to represent numeric and boolean expressions respectively. A ternary numeric expression $\mathbb{b}\ ?\ \mathbb{n}_1 : \mathbb{n}_2$ evaluates to $\mathbb{n}_1$ when the comparison evaluates to true, and $\mathbb{n}_2$ otherwise. Moreover, to model multiple queries to a database and produce multiple outputs during that process, ShadowDP supports lists: $e_1 :: e_2$ 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.

Random variables and expressions To model probabilistic computation, which is essential in differentially private algorithms, ShadowDP uses random variable $\eta \in \mathit{RVars}$ to store a sample drawn from a distribution. Random variables are similar to normal variables ($x \in \mathit{NVars}$) except that they are the only ones who can get random values from random expressions, via a sampling command $\eta := g$.

We follow the modular design of LightDP [42], where randomness expressions can be added easily. In this paper, we only consider the most interesting random expression, $\texttt{Lap}\ r$. Semantically, $\eta := \texttt{Lap}\ r$ draws one sample from the Laplace distribution, with mean zero and a scale factor r, and assigns it to η. For verification purpose, a sampling command also requires a few annotations, which we explain shortly.

Types Types in ShadowDP have the form of $\mathcal{B}_{\langle \mathbb{d}^\circ, \mathbb{d}^\dagger \rangle}$, where $\mathcal{B}$ is the base type, and $\mathbb{d}^\circ$, $\mathbb{d}^\dagger$ represent the alignments for the execution on adjacent database and shadow execution respectively. Base type is standard: it includes num (numeric type), bool (Boolean), or a list of elements with type τ (list τ).

Distance $\mathbb{d}$ is the key for randomness alignment proof. Intuitively, it relates two program executions so that the likelihood of seeing each is bounded by some constant. Since only numerical values have numeric distances, other data types (including bool, list τ and $\tau_1 \to \tau_2$) are always associated with $\langle 0, 0 \rangle$, hence omitted in the syntax. Note that this does not rule out numeric distances in nested types. For example, $(\texttt{list}\ \texttt{num}_{\langle 1,1 \rangle})$ stores numbers that differ by exactly one in both aligned and shadow executions.

Distance $\mathbb{d}$ can either be a numeric expression ($\mathbb{n}$) in the language or $*$. A variable x with type $\texttt{num}_{\langle *,* \rangle}$ is desugared as $x : \Sigma_{(\langle \widehat{x}^\circ : \texttt{num}_{\langle 0,0 \rangle}, \widehat{x}^\dagger : \texttt{num}_{\langle 0,0 \rangle} \rangle)}\ \texttt{num}_{\langle \widehat{x}^\circ, \widehat{x}^\dagger \rangle}$, where $\widehat{x}^\circ$, $\widehat{x}^\dagger$ are distinguished variables invisible in the source code; hiding those variables in a Σ-type simplifies the type system (Section 4).

The star type is useful for two reasons. First, it specifies the sensitivity of query answers in a precise way. Consider the parameter q in Figure 1 with type $\texttt{list}\ \texttt{num}_{\langle *,* \rangle}$, along with the precondition $\forall i \geq 0.\ -1 \leq \widehat{\texttt{q}}^\circ[\texttt{i}] \leq 1$. This notation makes the assumption of the Report Noisy Max algorithm explicit: each query answer differs by at most 1. Second, star type serves as the last resort when the distance of a variable cannot be tracked precisely by a static type system. For example, whenever ShadowDP merges two different distances (e.g., 3 and 4) of x from two branches, the result distance is $*$; the type system instruments the source code to maintain the correct values of $\widehat{x}^\circ, \widehat{x}^\dagger$ (Section 4).

Sampling with selectors Each sampling instruction is attached with a few annotations for proving differential privacy, in the form of $(\eta := \texttt{Lap}\ r, \mathcal{S}, \mathbb{n}_\eta)$. Note that just like types, the annotations $\mathcal{S}$, $\mathbb{n}_\eta$ have no effects on the program semantics; they only show up in verification. Intuitively, a selector $\mathcal{S}$ picks a version ($k \in \{\circ, \dagger\}$) for all program variables (including the previously sampled variables) at the sampling instruction, as well as constructs randomness alignment for η, specified by $\mathbb{n}_\eta$ (note that the distance cannot be $*$ by syntactical restriction here). By definition, both $\mathcal{S}$ and $\mathbb{n}_\eta$ may depend on the program state when the sampling happens.

Return to the running example in Figure 1. As illustrated in Figure 2, the selective alignment is to

- use shadow variables and align the new sample by 2 whenever a new max is encountered,
- use aligned variables and the same sample otherwise.

Hence, the sampling command in Figure 1 is annotated as $(\eta := \texttt{Lap}\ (2/\epsilon), \Omega\ ?\ \dagger : \circ, \Omega\ ?\ 2 : 0)$, where Ω is $\texttt{q[i]}+\eta > \texttt{bq} \vee \texttt{i} = 0$, the condition when a new max is found.

3.2 Semantics

As standard, 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 $m \in \mathcal{M}$ maps all (normal and random) variables ($NVars \cup RVars$) to their values.

The semantics of an expression e of base type $\mathcal{B}$ is interpreted as a function $[\![e]\!] : \mathcal{M} \to [\![\mathcal{B}]\!]$, where $[\![\mathcal{B}]\!]$ represents the set of values belonging to the base type $\mathcal{B}$. We omit expression semantics since it is standard. A random expression g is interpreted as a distribution on real values. Hence, $[\![g]\!] : \textbf{Dist}([\![\texttt{num}]\!])$. Moreover, a command c is interpreted as a function $[\![c]\!] : \mathcal{M} \to \textbf{Dist}(\mathcal{M})$. For brevity, we write $[\![e]\!]_m$ and $[\![c]\!]_m$ instead of $[\![e]\!](m)$ and $[\![c]\!](m)$ hereafter. Finally, all programs have the form $(c; \texttt{return}\ e)$ where c contains no return statement. A program is interpreted as a function $m \to \textbf{Dist}[\![\mathcal{B}]\!]$ where $\mathcal{B}$ is the return type (of e).

The semantics of commands is available in the full version of this paper [40]; the semantics directly follows a standard semantics in [28].

4 ShadowDP: Type System

ShadowDP is equipped with a flow-sensitive type system. If successful, it generates a transformed program with needed assertions to make the original program differentially private. The transformed program is simple enough to be verified by off-the-shelf program verifiers.

4.1 Notations

We denote by Γ the typing environment which tracks the type of each variable in a flow-sensitive way (i.e., the type of each variable at each program point is traced separately). All typing rules are formalized in Figure 4. Typing rules share a common global invariant Ψ, such as the sensitivity assumption annotated in the source code (e.g., the precondition in Figure 1). We also write $\Gamma(x) = \langle \mathbb{d}^\circ, \mathbb{d}^\dagger \rangle$ for $\exists\, \mathcal{B}.\ \Gamma(x) = \mathcal{B}_{\langle \mathbb{d}^\circ, \mathbb{d}^\dagger \rangle}$ when the base type $\mathcal{B}$ is irrelevant.

4.2 Expressions

Expression rules have the form of $\Gamma \vdash e : \tau$, which means that expression e has type τ under the environment Γ. Most rules are straightforward: they compute the distance for aligned and shadow executions separately. Rule (T-OTimes) makes a conservative approach for nonlinear computations, following LightDP [42]. Rule (T-VAR) desugars star types when needed. The most interesting rule is (T-ODot), which generates the following constraint:

$$\Psi \Rightarrow (e_1 \odot e_2 \Leftrightarrow (e_1 + \mathbb{n}_1) \odot (e_2 + \mathbb{n}_3) \wedge (e_1 + \mathbb{n}_2) \odot (e_2 + \mathbb{n}_4))$$

This constraint states that the boolean value of $e_1 \odot e_2$ is identical in both aligned and shadow executions. If the constraint is discharged by an external solver (our type system uses Z3 [16]), we are assured that $e_1 \odot e_2$ has distances $\langle 0, 0 \rangle$.

4.3 Commands

The flow-sensitive type system tracks and checks the distances of aligned and shadow executions at each program point. Typing rules for commands have the form of

$$pc \vdash \Gamma\ \{c \rightharpoonup c'\}\ \Gamma'$$

meaning that starting from the previous typing environment Γ, the new typing environment is Γ' after c. We will discuss the other components pc and c' shortly.

4.3.1 Aligned Variables

The type system infers and checks the distances of both aligned and shadow variables. Since most rules treat them in the same way, we first explain the rules with respect to aligned variables only, then we discuss shadow variables in Section 4.3.2. To simplify notation, we write Γ instead of Γ° for now since only aligned variables are discussed.

Flow-Sensitivity In each typing rule $pc \vdash \Gamma\ \{c \rightharpoonup c'\}\ \Gamma'$, an important invariant is that if c runs on two memories that are aligned by Γ, then the final memories are aligned by Γ'.

Consider the assignment rule (T-Asgn). This rule computes the distance of e's value, $\mathbb{n}^\circ$, and updates the distance of x's value after assignment to $\mathbb{n}^\circ$.

More interesting are rules (T-If) and (T-While). In (T-If), we compute the typing environments after executing c_1 and c_2 as Γ_1 and Γ_2 respectively. Since each branch may update x's distance in arbitrary way, $\Gamma_1(x)$ and $\Gamma_2(x)$ may differ. We note that numeric expressions and $*$ type naturally form a two level lattice, where $*$ is higher than any $\mathbb{n}$. Hence, we use the following rule to merge two distances $\mathbb{d}_1$ and $\mathbb{d}_2$:

$$\mathbb{d}_1 \sqcup \mathbb{d}_2 \triangleq \begin{cases} \mathbb{d}_1 & \text{if } \mathbb{d}_1 = \mathbb{d}_2 \\ * & \text{otherwise} \end{cases}$$

For example, $(3 \sqcup 4 = *)$, $(x + y \sqcup x + y = x + y)$, $(x \sqcup 3 = *)$. Hence, (T-If) ends with $\Gamma_1 \sqcup \Gamma_2$, defined as $\lambda x.\ \Gamma_1(x) \sqcup \Gamma_2(x)$.

As an optimization, we also use branch conditions to simplify distances. Consider our running example (Figure 1): at Line 4, η has (aligned) distance $\Omega\ ?\ 2 : 0$, where Ω is the branch condition. Its distance is simplified to 2 in the `true` branch and 0 in the `false` branch.

Rule (T-While) is similar, except that it requires a fixed point Γ_f such that $pc \vdash \Gamma \sqcup \Gamma_f\ \{c\}\ \Gamma_f$. In fact, this rule is deterministic since we can construct the fixed point as follows (the construction is similar to the one in [26]):

$$pc \vdash \Gamma'_i\ \{c \rightharpoonup c'_i\}\ \Gamma''_i \text{ for all } 0 \le i \le n$$

where $\Gamma'_0 = \Gamma, \Gamma'_{i+1} = \Gamma''_i \sqcup \Gamma, \Gamma'_{n+1} = \Gamma'_n$.

Typing rules for expressions

$$\frac{}{\Gamma \vdash r : \mathsf{num}_{\langle 0,0\rangle}}\ (\text{T-Num}) \qquad \frac{}{\Gamma \vdash b : \mathsf{bool}}\ (\text{T-Boolean}) \qquad \frac{\Gamma(x) = \mathcal{B}_{\langle \mathbb{d}^\circ, \mathbb{d}^\dagger\rangle} \quad \mathbb{n}^\star = \begin{cases} \widehat{x}^\star & \text{if } \mathbb{d}^\star = * \\ \mathbb{d}^\star & \text{otherwise} \end{cases} \quad \star \in \{\circ, \dagger\}}{\Gamma \vdash x : \mathcal{B}_{\langle \mathbb{n}^\circ, \mathbb{n}^\dagger\rangle}}\ (\text{T-Var})$$

$$\frac{\Gamma \vdash e_1 : \mathsf{num}_{\langle \mathbb{n}_1, \mathbb{n}_2\rangle} \quad \Gamma \vdash e_2 : \mathsf{num}_{\langle \mathbb{n}_3, \mathbb{n}_4\rangle}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{num}_{\langle \mathbb{n}_1 \oplus \mathbb{n}_3, \mathbb{n}_2 \oplus \mathbb{n}_4\rangle}}\ (\text{T-OPlus}) \qquad \frac{\Gamma \vdash e_1 : \mathsf{num}_{\langle 0,0\rangle} \quad \Gamma \vdash e_2 : \mathsf{num}_{\langle 0,0\rangle}}{\Gamma \vdash e_1 \otimes e_2 : \mathsf{num}_{\langle 0,0\rangle}}\ (\text{T-OTimes})$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathsf{num}_{\langle \mathbb{n}_1, \mathbb{n}_2\rangle} \\ \Gamma \vdash e_2 : \mathsf{num}_{\langle \mathbb{n}_3, \mathbb{n}_4\rangle}\end{array} \quad \begin{array}{r}\Psi \Rightarrow (e_1 \odot e_2 \Leftrightarrow (e_1 + \mathbb{n}_1) \odot (e_2 + \mathbb{n}_3)) \\ \wedge (e_1 \odot e_2 \Leftrightarrow (e_1 + \mathbb{n}_2) \odot (e_2 + \mathbb{n}_4))\end{array}}{\Gamma \vdash e_1 \odot e_2 : \mathsf{bool}}\ (\text{T-ODot}) \qquad \frac{\Gamma \vdash e_1 : \mathsf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1\ ?\ e_2 : e_3 : \tau}\ (\text{T-Ternary})$$

$$\frac{\Gamma \vdash e : \mathsf{bool}}{\Gamma \vdash \neg e : \mathsf{bool}}\ (\text{T-Neg}) \qquad \frac{\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \mathtt{list}\ \tau}{\Gamma \vdash e_1 :: e_2 : \mathtt{list}\ \tau}\ (\text{T-Cons}) \qquad \frac{\Gamma \vdash e_1 : \mathtt{list}\ \tau \quad \Gamma \vdash e_2 : \mathsf{num}_{\langle 0,0\rangle}}{\Gamma \vdash e_1[e_2] : \tau}\ (\text{T-Index})$$

Typing rules for commands

$$\frac{}{pc \vdash \Gamma\ \{\mathtt{skip} \rightharpoonup \mathtt{skip}\}\ \Gamma}\ (\text{T-Skip}) \qquad \frac{\Gamma \vdash e : \mathcal{B}_{\langle \mathbb{n}^\circ, \mathbb{n}^\dagger\rangle} \quad \langle \Gamma', c^\dagger\rangle = \begin{cases} \langle \Gamma[x \mapsto \mathcal{B}_{\langle \mathbb{n}^\circ, \mathbb{n}^\dagger\rangle}], \mathtt{skip}\rangle, & \text{if } pc = \bot \\ \langle \Gamma[x \mapsto \mathcal{B}_{\langle \mathbb{n}^\circ, *\rangle}], \widehat{x}^\dagger := x + \mathbb{n}^\dagger - e\rangle, & \text{else} \end{cases}}{pc \vdash \Gamma\ \{x := e \rightharpoonup c^\dagger; x := e\}\ \Gamma'}\ (\text{T-Asgn})$$

$$\frac{pc \vdash \Gamma\ \{c_1 \rightharpoonup c_1'\}\ \Gamma_1 \qquad pc \vdash \Gamma_1\ \{c_2 \rightharpoonup c_2'\}\ \Gamma_2}{pc \vdash \Gamma\ \{c_1; c_2 \rightharpoonup c_1'; c_2'\}\ \Gamma_2}\ (\text{T-Seq}) \qquad \frac{\Gamma \vdash e : \mathsf{num}_{\langle 0,\mathbb{d}\rangle} \quad or \quad \Gamma \vdash e : \mathsf{bool}}{pc \vdash \Gamma\ \{\mathtt{return}\ e \rightharpoonup \mathtt{return}\ e\}\ \Gamma}\ (\text{T-Return})$$

$$\frac{\begin{array}{l} pc' = \mathsf{updPC}(pc, \Gamma, e) \\ pc' \vdash \Gamma\ \{c_i \rightharpoonup c_i'\}\ \Gamma_i \end{array} \quad \Gamma_i, \Gamma_1 \sqcup \Gamma_2, pc' \Rrightarrow c_i''\ i \in \{1,2\} \quad c^\dagger = \begin{cases} \mathtt{skip}, & \text{if } (pc = \top \vee pc' = \bot) \\ (\!|\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \Gamma_1 \sqcup \Gamma_2|\!)^\dagger, & \text{else} \end{cases}}{pc \vdash \Gamma\ \left\{\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \rightharpoonup \left(\mathtt{if}\ e\ \mathtt{then}\ (\mathtt{assert}\ ((\!|e, \Gamma|\!)^\circ); c_1'; c_1'')\ \mathtt{else}\ (\mathtt{assert}\ (\neg(\!|e, \Gamma|\!)^\circ); c_2'; c_2'')\right); c^\dagger\right\}\ \Gamma_1 \sqcup \Gamma_2}\ (\text{T-If})$$

$$\frac{\begin{array}{l} pc' = \mathsf{updPC}(pc, \Gamma, e) \\ pc' \vdash \Gamma \sqcup \Gamma_f\ \{c \rightharpoonup c'\}\ \Gamma_f \end{array} \quad \begin{array}{l} \Gamma, \Gamma \sqcup \Gamma_f, pc' \Rrightarrow c_s \\ \Gamma_f, \Gamma \sqcup \Gamma_f, pc' \Rrightarrow c'' \end{array} \quad c^\dagger = \begin{cases} \mathtt{skip}, & \text{if } (pc = \top \vee pc' = \bot) \\ (\!|\mathtt{while}\ e\ \mathtt{do}\ c, \Gamma \sqcup \Gamma_f|\!)^\dagger, & \text{else} \end{cases}}{pc \vdash \Gamma\ \{\mathtt{while}\ e\ \mathtt{do}\ c \rightharpoonup c_s; (\mathtt{while}\ e\ \mathtt{do}\ (\mathtt{assert}\ ((\!|e, \Gamma|\!)^\circ); c'; c'')); c^\dagger\}\ \Gamma \sqcup \Gamma_f}\ (\text{T-While})$$

Typing rules for random assignments

$$\frac{pc = \bot \quad \Gamma' = \lambda x.\ \langle \mathcal{S}(\langle \mathbb{n}^\circ, \mathbb{n}^\dagger\rangle), \mathbb{n}^\dagger\rangle \text{ where } \Gamma \vdash x : \mathcal{B}_{\langle \mathbb{n}^\circ, \mathbb{n}^\dagger\rangle} \quad \Psi \Rightarrow ((\eta + \mathbb{n}_\eta)\{\eta_1/\eta\} = (\eta + \mathbb{n}_\eta)\{\eta_2/\eta\} \Rightarrow \eta_1 = \eta_2)}{pc \vdash \Gamma\ \{\eta := \mathsf{Lap}\ \ r; \mathcal{S}, \mathbb{n}_\eta \rightharpoonup \eta := \mathsf{Lap}\ \ r; \mathcal{S}, \mathbb{n}_\eta\}\ \Gamma'[\eta \mapsto \mathsf{num}_{\langle \mathbb{n}_\eta, 0\rangle}]}\ (\text{T-Laplace})$$

Instrumentation rule

$$\frac{\Gamma_1 \sqsubseteq \Gamma_2 \quad \begin{array}{l} c^\circ = \{\widehat{x}^\circ := \mathbb{n} \mid \Gamma_1(x) = \mathsf{num}_{\langle \mathbb{n}, \mathbb{d}_1\rangle} \wedge \Gamma_2(x) = \mathsf{num}_{\langle *, \mathbb{d}_2\rangle}\} \\ c^\dagger = \{\widehat{x}^\dagger := \mathbb{n} \mid \Gamma_1(x) = \mathsf{num}_{\langle \mathbb{d}_1, \mathbb{n}\rangle} \wedge \Gamma_2(x) = \mathsf{num}_{\langle \mathbb{d}_2, *\rangle}\} \end{array} \quad c' = \begin{cases} c^\circ; c^\dagger & \text{if } pc = \bot \\ c^\circ & \text{if } pc = \top \end{cases}}{\Gamma_1, \Gamma_2, pc \Rrightarrow c'}$$

Select function

$$\circ(\langle e_1, e_2\rangle) = e_1 \qquad \dagger(\langle e_1, e_2\rangle) = e_2 \qquad (e\ ?\ \mathcal{S}_1 : \mathcal{S}_2)(\langle e_1, e_2\rangle) = e\ ?\ \mathcal{S}_1(\langle e_1, e_2\rangle) : \mathcal{S}_2(\langle e_1, e_2\rangle)$$

PC update function

$$\mathsf{updPC}(pc, \Gamma, e) = \begin{cases} \bot\ , \text{if } pc = \bot \wedge \Psi \Rightarrow (e \Leftrightarrow (\!|e, \Gamma|\!)^\dagger) \\ \top\ , \text{else} \end{cases}$$

Figure 4. Typing rules and auxiliary rules. Ψ is an invariant that holds throughout program execution. In most rules, shadow distances are handled in the same way as aligned distances, with exceptions highlighted in gray boxes.

It is easy to check that $\Gamma'_n = \Gamma'_{n+1} = \Gamma''_n \sqcup \Gamma$ and $pc' \vdash \Gamma'_n \{c \rightharpoonup c'_i\} \Gamma''_n$ by construction. Hence, Γ''_n is a fixed point: $pc \vdash \Gamma \sqcup \Gamma''_n \{c \rightharpoonup c'_i\} \Gamma''_n$. Moreover, the computation above always terminates since all typing rules are monotonic on typing environments[2] and the lattice has a height of 2.

Maintaining dynamically tracked distances Each typing rule $pc \vdash \Gamma \{c \rightharpoonup c'\} \Gamma'$ also sets the value of $\widehat{x}^\circ$ to maintain distance dynamically whenever $\Gamma'(x) = *$. This is achieved by the instrumented commands in c'.

None of rules (T-Skip, T-Asgn, T-Seq, T-Ret) generate $*$ type, hence they do not need any instrumentation. The merge operation in rule (T-If) generates $*$ type when $\Gamma_1(x) \neq \Gamma_2(x)$. In this case, we use the auxiliary instrumentation rule in the form of $\Gamma_1, \Gamma_2, pc \Rrightarrow c'$, assuming $\Gamma_1 \sqsubseteq \Gamma_2$. In particular, for each variable x whose distance is "upgraded" to $*$, the rule sets $\widehat{x}^\circ$ to the distance previously tracked by the type system ($\Gamma_1(x)$). Moreover, the instrumentation commands c''_1, c''_2 are inserted under their corresponding branches.

Consider the following example:

```
if (x > 1) x := y; else y := 1;
```

staring with $\Gamma_0 : \{x : 1, y : 0\}$. In the `true` branch, rule (T-Asgn) updates x to the distance of y, resulting $\Gamma_1 : \{x : 0, y : 0\}$. Similarly, we get $\Gamma_2 : \{x : 1, y : 0\}$ in the `false` branch. Moreover, when we merge the typing environments Γ_1 and Γ_2 at the end of branch, the typing environment becomes $\Gamma_3 = \Gamma_1 \sqcup \Gamma_2 = \{x : *, y : 0\}$. Since $\Gamma_1(x) \neq \Gamma_2(x)$, instrumentation rule is also applied, which instruments $\widehat{\mathtt{x}}^\circ$ `:= 0` after `x :=` y and $\widehat{\mathtt{x}}^\circ$ `:= 1` after `y := 1`.

Rule (T-While) may also generate $*$ types. Following the same process in rule (T-If), it also uses the instrumentation rule to update corresponding dynamically tracked distance variables. The instrumentation command c_s is inserted before loop and c'' after the commands in the loop body.

Well-Formedness Whenever an assignment $x := e$ is executed, no variable's distance should depend on x. To see why, consider $x := 2$ with initial $\Gamma^\circ(y) = x$ and $m(x) = 1$. Since this assignment does not modify the value of y, the aligned value of y (i.e., $y + \Gamma^\circ(y)$) should not change. However, $\Gamma^\circ(y)$ changes from 1 to 2 after the assignment.

To avoid this issue, we check the following condition for each assignment $x := e$: $\forall y \in \mathit{Vars}.\ x \notin \mathit{Vars}(\Gamma(y))$. In case that the check fails for some y, we promote its distance to $*$, and use the auxiliary instrumentation $\Rrightarrow$ to set $\widehat{y}^\circ$ properly. Hence, *well-formedness* is guaranteed: no variable's distance depends on x when x is updated.

Aligned branches For differential privacy, we require the aligned execution to follow the same branch as the original execution. Due to dynamically tracked distances, statically checking that in a type system could be imprecise. Hence, we use assertions in rules (T-If) and (T-While) to ensure the aligned execution does not diverge. In those rules, $(\!|e, \Gamma|\!)^\circ$ simply computes the value of e in the aligned execution; its full definition is in the full version of this paper [40].

[2] That is, $\forall pc, c, \Gamma_1, \Gamma_2, \Gamma'_1, \Gamma'_2, c_1, c_2.\ pc \vdash \Gamma_i \{c \rightharpoonup c'_i\} \Gamma'_i\ i \in \{1, 2\} \wedge \Gamma_1 \sqsubseteq \Gamma_2 \implies \Gamma'_1 \sqsubseteq \Gamma'_2$.

4.3.2 Shadow Variables

In most typing rules, shadow variables are handled in the same way as aligned ones, which is discussed above. The key difference is that the type system allows the shadow execution to take a different branch from the original execution.

The extra permissiveness is the key ingredient of verifying algorithms such as Report Noisy Max. To see why, consider the example in Figure 2, where the shadow execution runs on D_2 with same random noise as from the execution on D_1. Upon the second query, the shadow execution does not update max, since its noisy value 3 is the same as the previous max; however, execution on D_1 will update max, since the noisy query value of 4 is greater than the previous max of 2.

To capture the potential divergence of shadow execution, each typing rule is associated with a program counter pc with two possible values $\bot$ and $\top$ (introducing program counters in a type system is common in information flow control to track implicit flows [36]). Here, $\top$ (resp. $\bot$) means that the shadow execution might take a different branch (resp. must take the same branch) as the original execution.

When $pc = \bot$, the shadow execution is checked in the same way as aligned execution. When $pc = \top$, the shadow distances are updated (as done in Rule (T-Asgn)) so that $x + \widehat{x}^\dagger$ remains the same. The new value from the shadow execution will be maintained by the type system *when pc transits from* $\bot$ *to* $\top$ by code instrumentation for sub-commands in (T-If) and (T-While), as we show next.

Take a branch (`if` e `then` c_1 `else` c_2) for example. The transition happens when $pc = \bot \wedge pc' = \top$. In this case, we construct a shadow execution of c by an auxiliary function $(\!|c, \Gamma|\!)^\dagger$. The shadow execution essentially replaces each variable x with their correspondence (i.e., $x + \widehat{x}^\dagger$), as is standard in self-composition [4, 38]. The only difference is that $(\!|c, \Gamma|\!)^\dagger$ is not applicable to sampling commands, since we are unable to align the sample variables when different amount of samples are taken. The full definition of $(\!|c, \Gamma|\!)^\dagger$ is available in the full version of this paper [40]. Rule (T-While) is very similar in its way of handling shadow variables.

4.3.3 Sampling Command

Rule (T-Laplace) checks the only probabilistic command $\eta :=$ `Lap` $r, \mathcal{S}, \mathbb{n}_\eta$ in ShadowDP. Here, the selector $\mathcal{S}$ and numeric distance $\mathbb{n}_\eta$ are annotations provided by a programmer to aid type checking. For the sample η, the aligned distance is specified by $\mathbb{n}_\eta$ and the shadow distance is always 0 (since by definition, shadow execution use the same sample as the original program). Hence, the type of η becomes $\mathtt{num}_{\langle \mathbb{n}_\eta, 0 \rangle}$.

Moreover, the selector constructs the aligned execution from either the aligned ($\circ$) or shadow ($\dagger$) execution. Since

the selector may depend on a condition e, we use the selector function $\mathcal{S}(\langle e_1, e_2\rangle)$ in Figure 4 to do so.

Rule (T-Laplace) also checks that each η is generated in an injective way: the same aligned value of η implies the same value of η in the original execution.

Consider the sampling command in Figure 1. The typing environments before and after the command is shown below (we omit unrelated parts for brevity):

$$\{\mathsf{bq} : \langle *, * \rangle, \cdots\}$$
$$\eta \ := \ \mathsf{Lap}\ (2/\epsilon), \Omega\ ?\ \dagger : \circ, \Omega\ ?\ 2 : 0;$$
$$\{\mathsf{bq} : \langle \Omega\ ?\ \widehat{\mathsf{bq}}^{\dagger} : \widehat{\mathsf{bq}}^{\circ}, \widehat{\mathsf{bq}}^{\dagger} \rangle, \eta : \langle \Omega\ ?\ 2 : 0, 0 \rangle, \cdots\}$$

In this example, $\mathcal{S}$ is $\Omega\ ?\ \dagger : \circ$. So the aligned distance of variable bq will be $\Omega\ ?\ \widehat{\mathsf{bq}}^{\dagger} : \widehat{\mathsf{bq}}^{\circ}$, the shadow distance of variable bq is still $\widehat{\mathsf{bq}}^{\dagger}$. The aligned distance of η is $\langle \Omega\ ?\ 2 : 0, 0\rangle$, where the aligned part is specified in the annotation.

4.4 Target Language

One goal of ShadowDP is to enable verification of ϵ-differential privacy using off-the-shelf verification tools. In the transformed code so far, we assumed `assert` commands to verify that certain condition holds. The only remaining challenging feature is the sampling commands, which requires probabilistic reasoning. Motivated by LightDP [42], we note that for ϵ-differential privacy, we are only concerned with the maximum privacy cost, not its likelihood. Hence, in the final step, we simply replace the sampling command with a non-deterministic command havoc η, which semantically sets the variable η to an arbitrary value upon execution, as shown in Figure 5.

Note that a distinguished variable $\mathbf{v}_\epsilon$ is added by the type system to explicitly track the privacy cost of the original program. For Laplace distribution, aligning η by the distance of $\mathbb{n}_\eta$ is associated with a privacy cost of $|\mathbb{n}_\eta|/r$. The reason is that the ratio of any two points that are $|\mathbb{n}_\eta|$ apart in the Laplace distribution with scaling factor r is bounded by $\exp(|\mathbb{n}_\eta|/r)$. Since the shadow execution uses the same sample, it has no privacy cost. This very fact allows us to *reset* privacy cost when the shadow execution is used (i.e., $\mathcal{S}$ selects †): the rule sets privacy cost to $0 + |\mathbb{n}_\eta|/r$ in this case.

In Figure 1, $\mathbf{v}_\epsilon$ is set to $\Omega\ ?\ 0 : \mathbf{v}_\epsilon + \Omega\ ?\ \epsilon : 0$ which is the same as $\Omega\ ?\ \epsilon : \mathbf{v}_\epsilon$. Intuitively, that implies that the privacy cost of the entire algorithm is either ϵ (when a new max is found) or the same as the previous value of $\mathbf{v}_\epsilon$.

The type system guarantees the following important property: if the original program type checks and the privacy cost $\mathbf{v}_\epsilon$ in the target language is bounded by some constant ϵ in all possible executions of the program, then the original program satisfies ϵ-differential privacy. We will provide a soundness proof in the next section. Consider the running example in Figure 1. The transformed program in the target language is shown at the bottom. With a model checking tool CPAChecker [11], we verified that $\mathbf{v}_\epsilon \le \epsilon$ in the transformed program within 2 seconds (Section 6.3). Hence, the Report Noisy Max algorithm is verified to be ϵ-differentially private.

$$\frac{}{\eta := \mathsf{Lap}\ \ r; \mathcal{S}, \mathbb{n}_\eta \rightrightarrows \mathsf{havoc}\ \ \eta; \mathbf{v}_\epsilon := \mathcal{S}(\langle \mathbf{v}_\epsilon, 0\rangle) + |\mathbb{n}_\eta|/r;}$$

$$\frac{}{c \rightrightarrows c, \text{ if } c \text{ is not a sampling command}}$$

Figure 5. Transformation rules to the target language. Probabilistic commands are reduced to non-deterministic ones.

5 Soundness

The type system performs a two-stage transformation:

$$pc \vdash \Gamma_1\ \{c \rightharpoonup c'\}\ \Gamma_2 \quad \text{and} \quad c' \rightrightarrows c''$$

Here, both c and c' are probabilistic programs; the difference is that c executes on the original memory without any distance tracking variables; c' executes on the extended memory where distance tracking variables are visible. In the second stage, c' is transformed to a non-probabilistic program c'' where sampling instructions are replaced by havoc and the privacy cost $\mathbf{v}_\epsilon$ is explicit. In this section, we use c, c', c'' to represent the source, transformed, and target program respectively.

Overall, the type system ensures ϵ-differential privacy (Theorem 2): if the value of $\mathbf{v}_\epsilon$ in c'' is always bounded by a constant ϵ, then c is ϵ-differentially private. In this section, we formalize the key properties of our type system and prove its soundness. Due to space constraints, the complete proofs are available in the full version of this paper [40].

Extended Memory Command c' is different from c since it maintains and uses distance tracking variables. To close the gap, we first extend memory m to include those variables, denoted as $\widehat{Vars} = \cup_{x \in NVars}\{\widehat{x}^\circ, \widehat{x}^\dagger\}$ and introduce a distance environment $\gamma : \widehat{Vars} \to \mathbb{R}$.

Definition 2. *Let $\gamma : \widehat{\mathrm{Vars}} \to \mathbb{R}$. For any $m \in \mathcal{M}$, there is an extension of m,written $m \uplus (\gamma)$, such that*

$$m \uplus (\gamma)(x) = \begin{cases} m(x), & x \in \mathrm{Vars} \\ \gamma(x), & x \in \widehat{\mathrm{Vars}} \end{cases}$$

We use $\mathcal{M}'$ to denote the set of extended memory states and m_1', m_2' to refer to concrete extended memory states. We note that although the programs c and c' are probabilistic, the extra commands in c' are deterministic. Hence, c' preserves the semantics of c, as formalized by the following Lemma.

Lemma 1 (Consistency)**.** *Suppose $pc \vdash \Gamma_1\ \{c \rightharpoonup c'\}\ \Gamma_2$. Then for any initial and final memory m_1, m_2 such that $[\![c]\!]_{m_1}(m_2) \neq 0$, and any extension m_1' of m_1, there is a unique extension m_2' of m_2 such that*

$$[\![c']\!]_{m_1'}(m_2') = [\![c]\!]_{m_1}(m_2)$$

Proof. By structural induction on c. The only interesting case is the (probabilistic) sampling command, which does not modify distance tracking variables. □

From now on, we will use m_2' to denote the unique extension of m_2 satisfying the property above.

Γ-*Relation* To formalize and prove the soundness property, we notice that a typing environment Γ along with distance environment γ induces two binary relations on memories. We write $m_1 \uplus (\gamma)\ \Gamma^\circ\ m_2$ (resp. $m_1 \uplus (\gamma)\ \Gamma^\dagger\ m_2$) when m_1, m_2 are related by Γ° (resp. $\Gamma^\dagger$) and γ. Intuitively, the initial γ and Γ (given by the function signature) specify the adjacency relation, and the relation is maintained by the type system throughout program execution. For example, the initial γ and Γ in Figure 1 specifies that two executions of the program is related if non-private variables ϵ, *size* are identical, and each query answer in $q[i]$ differs by at most one.

To facilitate the proof, we simply write $m_1'\ \Gamma\ m_2$ where m_1' is an extended memory in the form of $m_1 \uplus (\gamma)$.

Definition 3 (Γ-Relations)**.** *Two memories m_1' (in the form of $m_1 \uplus (\gamma)$) and m_2 are related by Γ°, written $m_1'\ \Gamma^\circ\ m_2$, if $\forall x \in \text{Vars} \cup \text{RVars}$, we have*

$$m_2(x) = m_1'(x) + m_1'(\mathbb{d}^\circ) \text{ if } \Gamma \vdash x : \mathsf{num}_{\langle \mathbb{d}^\circ, \mathbb{d}^\dagger \rangle}$$

We define the relation on non-numerical types and the $\Gamma^\dagger$ relation in a similar way.

By the definition above, Γ° introduces a function from $\mathcal{M}'$ to $\mathcal{M}$. Hence, we use $\Gamma^\circ m_1'$ as the unique m_2 such that $m_1'\ \Gamma^\circ\ m_2$. The $\Gamma^\dagger$ counterparts are defined similarly.

Injectivity For alignment-based proofs, given any γ, both Γ° and $\Gamma^\dagger$ must be injective functions [42]. The injectivity of Γ over the entire memory follows from the injectivity of Γ over the random noises $\eta \in$ *RVars*, which is checked as the following requirement in Rule (T-Laplace):

$$\Psi \Rightarrow ((\eta + \mathbb{n}_\eta)\{\eta_1/\eta\} = (\eta + \mathbb{n}_\eta)\{\eta_2/\eta\} \Rightarrow \eta_1 = \eta_2)$$

where all variables are universally quantified. Intuitively, this is true since the non-determinism of the program is purely from that of $\eta \in$ *RVars*.

Lemma 2 (Injectivity)**.** *Given $c, c', pc, m', m_1', m_2', \Gamma_1, \Gamma_2$ such that $pc \vdash \Gamma_1\{c \rightharpoonup c'\}\Gamma_2$, $[\![c']\!]_{m'} m_1' \neq 0 \wedge [\![c']\!]_{m'} m_2' \neq 0, \star \in \{\circ, \dagger\}$, then we have*

$$\Gamma_2^\star m_1' = \Gamma_2^\star m_2' \implies m_1' = m_2'$$

Soundness The soundness theorem connects the "privacy cost" of the probabilistic program to the distinguished variable $\mathbf{v}_\epsilon$ in the target program c''. To formalize the connection, we first extend memory one more time to include $\mathbf{v}_\epsilon$:

Definition 4. *For any extended memory m' and constant ϵ, there is an extension of m', written $m' \uplus (\epsilon)$, so that*

$$m' \uplus (\epsilon)(\mathbf{v}_\epsilon) = \epsilon, \text{ and } m' \uplus (\epsilon)(x) = m(x),\ \forall x \in \mathsf{dom}(m').$$

For a transformed program and a pair of initial and final memories m_1' and m_2', 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_1' and m_2' respectively:

Definition 5. *Given $c' \rightrightarrows c''$, m_1' and m_2', the consistent costs of executing c'' w.r.t. m_1' and m_2', written $c'' \restriction_{m_1'}^{m_2'}$, is defined as*

$$c'' \restriction_{m_1'}^{m_2'} \triangleq \{\epsilon \mid m_2' \uplus (\epsilon) \in [\![c'']\!]_{m_1' \uplus (0)}\}$$

Since $(c'' \restriction_{m_1'}^{m_2'})$ by definition is a set of values of $\mathbf{v}_\epsilon$, we write $\max(c'' \restriction_{m_1'}^{m_2'})$ for the maximum cost.

The next lemma enables precise reasoning of privacy cost w.r.t. a pair of initial and final memories:

Lemma 3 (Pointwise Soundness)**.** *Let $pc, c, c', c'', \Gamma_1, \Gamma_2$ be such that $pc \vdash \Gamma_1\{c \rightharpoonup c'\}\Gamma_2 \wedge c' \rightrightarrows c''$, then $\forall m_1', m_2'$:*

(i) the following holds:

$$[\![c']\!]_{m_1'}(m_2') \leq [\![c]\!]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\dagger m_2') \text{ when } pc = \bot \tag{1}$$

(ii) one of the following holds:

$$[\![c']\!]_{m_1'}(m_2') \leq \exp(\max(c'' \restriction_{m_1'}^{m_2'}))[\![c]\!]_{\Gamma_1^\circ m_1'}(\Gamma_2^\circ m_2') \tag{2a}$$

$$[\![c']\!]_{m_1'}(m_2') \leq \exp(\max(c'' \restriction_{m_1'}^{m_2'}))[\![c]\!]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\circ m_2') \tag{2b}$$

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 6. *For any program c'' in the target language, we say the execution cost of c'' is* bounded *by some constants ϵ, written $c''^{\leq \epsilon}$, iff for any m_1', m_2',*

$$m_2' \uplus (\epsilon') \in [\![c'']\!]_{m_1' \uplus (0)} \Rightarrow \epsilon' \leq \epsilon$$

Note that off-the-shelf tools can be used to verify that $c''^{\leq \epsilon}$ holds for some ϵ.

Theorem 1 (Soundness)**.** *Given $c, c', c'', m_1', \Gamma_1, \Gamma_2, \epsilon$ such that $\bot \vdash \Gamma_1\{c \rightharpoonup c'\}\Gamma_2 \wedge c' \rightrightarrows c'' \wedge c''^{\leq \epsilon}$, one of the following holds:*

$$\max_{S \subseteq \mathcal{M}'} ([\![c']\!]_{m_1'}(S) - \exp(\epsilon)[\![c]\!]_{\Gamma_1^\circ m_1'}(\Gamma_2^\circ S)) \leq 0, \tag{3a}$$

$$\max_{S \subseteq \mathcal{M}'} ([\![c']\!]_{m_1'}(S) - \exp(\epsilon)[\![c]\!]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\circ S)) \leq 0. \tag{3b}$$

Proof. By definition of $c''^{\leq \epsilon}$, we have $\max(c'' \restriction_{m_1'}^{m_2'}) \leq \epsilon$ for all $m_2' \in S$. Thus, by Lemma 3, we have one of the two:

$$[\![c']\!]_{m_1'}(m_2') \leq \exp(\epsilon)[\![c]\!]_{\Gamma_1^\circ m_1'}(\Gamma_2^\circ m_2'), \quad \forall m_2' \in S,$$

$$[\![c']\!]_{m_1'}(m_2') \leq \exp(\epsilon)[\![c]\!]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\circ m_2'), \quad \forall m_2' \in S.$$

If the first inequality is true, then

$$\max_{S \subseteq \mathcal{M}'} ([\![c']\!]_{m_1'}(S) - \exp(\epsilon)[\![c]\!]_{\Gamma_1^\circ m_1'}(\Gamma_2^\circ S))$$

$$= \max_{S \subseteq \mathcal{M}'} \sum_{m_2' \in S} ([\![c']\!]_{m_1'}(m_2') - \exp(\epsilon)[\![c]\!]_{\Gamma_1^\circ m_1'}(\Gamma_2^\circ m_2')) \leq 0$$

and therefore (3a) holds. Similarly, (3b) holds if the second inequality is true. Note that the equality above holds due to the injective assumption, which allows us to derive the set-based privacy from the point-wise privacy (Lemma 3). □

We now prove the main theorem on differential privacy:

Theorem 2 (Privacy). *Given* $\Gamma_1, \Gamma_2, c, c', c'', e, \epsilon$ *such that* $\Gamma_1^\circ = \Gamma_1^\dagger \wedge \bot \vdash \Gamma_1\{(c; \texttt{return } e) \rightharpoonup (c'; \texttt{return } e)\}\Gamma_2 \wedge c' \rightrightarrows c''$, *we have*

$$c''^{\leq \epsilon} \Rightarrow c \text{ is } \epsilon\text{-differentially private.}$$

Proof. By the typing rule, we have $\bot \vdash \Gamma_1\{c \rightharpoonup c'\}\Gamma_2$. By the soundness theorem (Theorem 1) and the fact that $\Gamma_1^\circ = \Gamma_2^\dagger$, we have $[\![c']\!]_{m_1'}(S) \leq \exp(\epsilon)[\![c]\!]_{\Gamma_1^\circ m_1'}(\Gamma_2^\circ S)$. For clarity, we stress that all sets are over distinct elements (as we have assumed throughout this paper).

By rule (T-Return), $\Gamma_2 \vdash e : \texttt{num}_{\langle 0, \mathrm{d}\rangle}$ or $\Gamma_2 \vdash e : \texttt{bool}$. For any set of values $V \subseteq [\![\mathcal{B}]\!]$, let $S'_V = \{m' \in \mathcal{M}' \mid [\![e]\!]_{m'} \in V\}$ and $S_V = \{m \in \mathcal{M} \mid [\![e]\!]_m \in V\}$, then we have $\Gamma_2^\circ S'_V \subseteq S_V$:

$$\begin{aligned} m \in \Gamma_2^\circ S'_V &\Rightarrow m = \Gamma_2^\circ m' \text{ for some } m' \in S_V \\ &\Rightarrow [\![e]\!]_m = [\![e]\!]_{\Gamma_2^\circ m'} = [\![e]\!]_{m'} \in V \\ &\Rightarrow m \in S_V. \end{aligned}$$

The equality in second implication is due to the zero distance when $\Gamma_2 \vdash e : \texttt{num}_{\langle 0, \mathrm{n}\rangle}$, and rule (T-Odot) when $\Gamma_2 \vdash e : \texttt{bool}$. We note that $\Gamma_2^\circ S'_V \neq S_V$ in general since Γ_2° might not be a surjection. Let $P' = (c'; \texttt{return } e)$, then for any γ, we have

$$\begin{aligned} [\![P']\!]_{m_1 \uplus (\gamma)}(V) &= [\![c']\!]_{m_1 \uplus (\gamma)}(S'_V) \\ &\leq \exp(\epsilon)[\![c']\!]_{\Gamma_1^\circ m_1 \uplus (\gamma)}(\Gamma_2^\circ S'_V) \\ &\leq \exp(\epsilon)[\![c']\!]_{\Gamma_1^\circ m_1 \uplus (\gamma)}(S_V) \\ &= \exp(\epsilon)[\![P]\!]_{\Gamma_1^\circ m_1 \uplus (\gamma)}(V). \end{aligned}$$

Finally, due to Lemma 1, $[\![P]\!]_{m_1}(V) = [\![P']\!]_{m_1 \uplus (\gamma)}(V)$. Therefore, by definition of privacy c is ϵ-differentially private. □

Note that the shallow distances are only useful for proofs; they are irrelevant to the differential privacy property being obeyed by a program. Hence, initially, we have $\Gamma_1^\circ = \Gamma_1^\dagger$ (both describing the adjacency requirement) in Theorem 2, as well as in all of the examples formally verified by ShadowDP.

6 Implementation and Evaluation

6.1 Implementation

We have implemented ShadowDP into a trans-compiler[3] in Python. ShadowDP currently supports trans-compilation from annotated C code to target C code. Its workflow includes two phases: *transformation* and *verification*. The annotated source code will be checked and transformed by ShadowDP; the transformed code is further sent to a verifier.

[3]Publicly available at https://github.com/cmla-psu/shadowdp.

Transformation As explained in Section 4, ShadowDP tracks the typing environments in a flow-sensitive way, and instruments corresponding statements when appropriate. Moreover, ShadowDP adds an assertion `assert` ($\mathbf{v}_\epsilon \leq \epsilon$) before the `return` command. This assertion specifies the final goal of proving differential privacy. The implementation follows the typing rules explained in Section 4.

Verification The goal of verification is to prove the assertion `assert` ($\mathbf{v}_\epsilon \leq \epsilon$) never fails for any possible inputs that satisfy the precondition (i.e., the adjacency requirement). To demonstrate the usefulness of the transformed programs, we use a model checker CPAChecker [11] v1.8. CPAChecker is capable of automatically verifying C program with a given configuration. In our implementation, *predicate analysis* is used. Also, CPAChecker has multiple solver backends such as MathSat [15], Z3 [16] and SMTInterpol [14]. For the best performance, we concurrently use different solvers and return the results as soon as any one of them verifies the program.

One limitation of CPAChecker and many other tools, is the limited support for non-linear arithmetics. For programs with non-linear arithmetics, we take two approaches. First, we verify the algorithm variants where ϵ is fixed (the approach taken in [2]). In this case, all transformed code in our evaluation is directly verified without any modification. Second, to verify the correctness of algorithms with arbitrary ϵ, we slightly rewrite the non-linear part in a linear way or provide loop invariants (see Section 6.2.2). We report the results from both cases whenever we encounter this issue.

6.2 Case Studies

We investigate some interesting differentially private algorithms that are formally verified by ShadowDP. We only present the most interesting programs in this section; the rest are provided in the full version of this paper [40].

6.2.1 Sparse Vector Technique

Sparse Vector Technique [20] is a powerful mechanism which has been proven to satisfy ϵ-differential privacy (its proof is notoriously tricky to write manually [29]). In this section we show how ShadowDP verifies this algorithm and later show how a novel variant is verified.

Figure 6 shows the pseudo code of Sparse Vector Technique [20]. It examines the input queries and reports whether each query is above or below a threshold T. To achieve differential privacy, it first adds Laplace noise to the threshold T, compares the noisy query answer $q[i] + \eta_2$ with the noisy threshold $\tilde{T}$, and returns the result (`true` or `false`). The number of `true`'s the algorithm can output is bounded by argument N. One key observation is that once the noise has been added to the threshold, outputting `false` pays no privacy cost [20]. As shown in Figure 6, programmers only have to provide two simple annotations: $\circ$, 1 for η_1 and $\circ$, Ω ? 2 : 0 for η_2. Since the selectors in this example only

Table 1. Time spent on type checking and verification

Algorithm	Type Check (s)	Verification by ShadowDP (s)		Verification by [2] (s)
Report Noisy Max	0.465	1.932		22
Sparse Vector Technique ($N = 1$)	0.398	1.856		27
		Rewrite	**Fix ϵ**	
Sparse Vector Technique	0.399	2.629	1.679	580
Numerical Sparse Vector Technique ($N = 1$)	0.418	1.783	1.788	4
Numerical Sparse Vector Technique	0.421	2.584	1.662	5
Gap Sparse Vector Technique	0.424	2.494	1.826	N/A
Partial Sum	0.445	1.922	1.897	14
Prefix Sum	0.449	1.903	1.825	14
Smart Sum	0.603	2.603	2.455	255

function SVT (ϵ, size, T, N : num$_{\langle 0,0\rangle}$; q : list num$_{\langle *,*\rangle}$)
returns (out : list bool)
precondition $\forall i \geq 0.\ -1 \leq \widehat{q}^{\circ}[i] \leq 1 \wedge \widehat{q}^{\dagger}[i] = \widehat{q}^{\circ}[i]$

```
1  η1 := Lap (2/ε), ∘, 1 ;
2  T̃ := T + η1; count := 0; i := 0;
3  while (count < N ∧ i < size)
4    η2 := Lap (4N/ε), ∘, Ω ? 2 : 0 ;
5    if (q[i] + η2 ≥ T̃) then
6      out := true::out;
7      count := count + 1;
8    else
9      out := false::out;
10   i := i + 1;
```

The transformed program (slightly simplified for readability), where underlined commands are added by the type system:

```
1  v_ε := 0;
2  havoc η1; v_ε := v_ε + ε/2;
3  T̃ := T + η1; count:= 0; i := 0;
4  while (count < N ∧ i < size)
5    assert (count < N ∧ i < size);
6    havoc η2; v_ε = Ω ? (v_ε + 2 × ε/4N) : (v_ε + 0);
7    if (q[i] + η2 ≥ T̃) then
8      assert (q[i] + q̂°[i] + η2 + 2 ≥ T̃ + 1);
9      out := true::out;
10     count := count + 1;
11   else
12     assert (¬(q[i] + q̂°[i] + η2 ≥ T̃ + 1));
13     out := false::out;
14   i := i + 1;
```

Figure 6. Verifying Sparse Vector Technique with ShadowDP (slightly simplified for readability). Annotations are in gray where Ω represents the branch condition.

select aligned version of variables, the shadow execution is optimized away (controlled by *pc* in rule (T-If)). ShadowDP successfully type checks and transforms this algorithm. However, due to a nonlinear loop invariant that CPAChecker fails to infer, it fails to verify the program. With the loop invariant provided manually, the verification succeeds, proving this algorithm satisfies ϵ-differential privacy (we also verified a variant where ϵ is fixed to N to remove the non-linearity).

6.2.2 Gap Sparse Vector Technique

We now consider a novel variant of Sparse Vector Technique. In this variant, whenever $q[i] + \eta_2 \geq \tilde{T}$, it outputs the value of the gap $q[i] + \eta_2 - \tilde{T}$ (how much larger the noisy answer is compared to the noisy threshold). Note that the noisy query value $q[i] + \eta_2$ is reused for both this check and the output (whereas other proposals either (1) draw fresh noise and result in a larger ϵ [20], or (2) re-use the noise but do not satisfy differential privacy, as noted in [29]). For noisy query values below the noisy threshold, it only outputs `false`. We call this algorithm GapSparseVector. More specifically, Line 6 in Figure 6 is changed from `out := true::out;` to the following: `out := (q[i] + η2 - T̃)::out;`. To the best of our knowledge, the correctness of this variant has not been noticed before. This variant can be easily verified with little changes to the original annotation. One observation is that, to align the `out` variable, the gap appended to the list must have 0 aligned distance. Thus we change the distance of η_2 from $\Omega\ ?\ 2 : 0$ to $\Omega\ ?\ (1 - \widehat{q}^{\circ}[i]) : 0$, the other part of the annotation remains the same.

ShadowDP successfully type checks and transforms the program. Due to the non-linear arithmetics issue, we rewrite the assignment command $\mathbf{v}_\epsilon := \mathbf{v}_\epsilon + (1 - \widehat{q}^{\circ}[i]) \times \epsilon/4N$; to **assert** $(|1 - \widehat{q}^{\circ}[i]| \leq 2)$; $\mathbf{v}_\epsilon := \mathbf{v}_\epsilon + 2 \times \epsilon/4N$; and provide nonlinear loop invariants; then it is verified (we also verified a variant where ϵ is fixed to 1).

6.3 Experiments

ShadowDP is evaluated on Report Noisy Max algorithm (Figure 1) along with all the algorithms discussed in Section 6.2, as well as Partial Sum, Prefix Sum and Smart Sum algorithms that are included in the full version of this paper [40]. For comparison, all the algorithms verified in [2] are included in the experiments (where Sparse Vector Technique is called Above Threshold in [2]). One exception is `ExpMech` algorithm, since ShadowDP currently lacks a sampling command

for `Exponential` noise. However, as shown in [42], it should be fairly easy to add a noise distribution without affecting the rest of a type system.

Experiments are performed on a Dual Intel® Xeon® E5-2620 v4@2.10GHz CPU machine with 64 GB memory. All algorithms are successfully checked and transformed by ShadowDP and verified by CPAChecker. For programs with non-linear arithmetics, we performed experiments on both solutions discussed in Section 6.2.2. Transformation and verification all finish within 3 seconds, as shown in Table 1, indicating the simplicity of analyzing the transformed program, as well as the practicality of verifying ϵ-differentially private algorithms with ShadowDP.

6.4 Proof Automation

ShadowDP requires two kinds of annotations: (1) function specification and (2) annotation for sampling commands. As most verification tools, (1) is required since it specifies the property being verified. In all of our verified examples, (2) is fairly simple and easy to write. To further improve the usability of ShadowDP, we discuss some heuristics to automatically generate the annotations for sampling commands.

Sampling commands requires two parts of annotation:

1. **Selectors**. The selector has two options: aligned ($\circ$) or shadow ($\dagger$), with potential dependence. The heuristic is to enumerate branch conditions. For Report Noisy Max, there is only one branch condition Ω, giving us four possibilities: $\circ$ / $\dagger$ / $\Omega\ ?\ \circ : \dagger$ / $\Omega\ ?\ \dagger : \circ$.
2. **Alignments for the sample**. It is often simple arithmetic on a small integer such as 0, 1, 2 or the exact difference of query answers and other program variables. For dependent types, we can also use the heuristic of using branch conditions. For Report Noisy Max, this will discover the correct alignment $\Omega\ ?\ 2 : 0$.

This enables the discovery of all the correct annotations for the algorithm studied in this paper. We leave a systematic study of proof automation as future work.

7 Related Work

Randomness alignment based proofs The most related work is LightDP [42]. ShadowDP is inspired by LightDP in a few aspects, but also with three significant differences. First, ShadowDP supports shadow execution, a key enabling technique for the verification of Report Noisy Max based on standard program semantics. Second, while LightDP has a flow-insensitive type system, ShadowDP is equipped with a flow-sensitive one. The benefit is that the resulting type system is both more expressive and more usable, since only sampling command need annotations. Third, ShadowDP allows extra permissiveness of allowing two related executions to take different branches, which is also crucial in verifying Report Noisy Max. In fact, ShadowDP is strictly more expressive than LightDP: LightDP is a restricted form of ShadowDP where the shadow execution is never used (i.e., when the selector always picks the aligned execution).

Coupling based proofs The state-of-the-art verifier based on approximate coupling [2] is also able to verify the algorithms we have discussed in this paper. Notably, it is able to automatically verify proofs for algorithms including Report-Noisy-Max and Sparse Vector. However, verifying the transformed program by ShadowDP is significantly easier than verifying the first-order Horn clauses and probabilistic constraints generated by their tool. In fact, ShadowDP verifies all algorithms within 3 seconds while the coupling verifier takes 255 seconds in verifying Smart Sum and 580 seconds in verifying Sparse Vector (excluding proof synthesis time). Also, instead of building the system on *customized* relational logics to verify differential privacy [3, 5, 8–10], ShadowDP bases itself on *standard* program logics, which makes the transformed program re-usable by other program analyses.

Other language-based proofs Recent work such as Personalized Differential Privacy (PDP) [21] allows each individual to set its own different privacy level and PDP will satisfy difference privacy regarding the level she sets. PINQ [31] tracks privacy consumption dynamically on databases and terminate when the privacy budget is exhausted. However, along with other work such as computing bisimulations families for probabilistic automata [39, 41], they fail to provide a tight bound on the privacy cost of sophisticated algorithms.

8 Conclusions and Future Work

In this paper we presented ShadowDP, a new language for the verification of differential privacy algorithms. ShadowDP uses shadow execution to generate more flexible randomness alignments that allows it to verify more algorithms, such as Report Noisy Max, than previous work based on randomness alignments. We also used it to verify a novel variant of Sparse Vector that reports the gap between noisy above-threshold queries and the noisy threshold.

Although ShadowDP only involves minimum annotations, one future work is to fully automate the verification using ShadowDP, as sketched in Section 6.4. Another natural next step is to extend ShadowDP to support more noise distributions, enabling it to verify more algorithms such as `ExpMech` which uses `Exponential` noise. Furthermore, we plan to investigate other applications of the transformed program. For instance, applying symbolic executors and bug finding tools on the transformed program to construct counterexamples when the original program is buggy.

Acknowledgments

We thank our shepherd Dana Drachsler-Cohen and anonymous PLDI reviewers for their helpful suggestions. This work is funded by NSF awards #1228669, #1702760, #1816282 and #1566411.

References

[1] John M. Abowd. 2018. The U.S. Census Bureau Adopts Differential Privacy. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '18)*. ACM, New York, NY, USA, 2867–2867.

[2] Aws Albarghouthi and Justin Hsu. 2017. Synthesizing Coupling Proofs of Differential Privacy. *Proceedings of ACM Programming Languages* 2, POPL, Article 58 (Dec. 2017), 30 pages.

[3] Gilles Barthe, George Danezis, Benjamin Gregoire, Cesar Kunz, and Santiago Zanella-Beguelin. 2013. Verified Computational Differential Privacy with Applications to Smart Metering. In *Proceedings of the 2013 IEEE 26th Computer Security Foundations Symposium (CSF '13)*. IEEE Computer Society, Washington, DC, USA, 287–301.

[4] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. 2004. Secure Information Flow by Self-Composition. In *Proceedings of the 17th IEEE Workshop on Computer Security Foundations (CSFW '04)*. IEEE Computer Society, Washington, DC, USA, 100–.

[5] Gilles Barthe, Noémie Fong, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Advanced Probabilistic Couplings for Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 55–67.

[6] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. 2014. Proving Differential Privacy in Hoare Logic. In *Proceedings of the 2014 IEEE 27th Computer Security Foundations Symposium (CSF '14)*. IEEE Computer Society, Washington, DC, USA, 411–424.

[7] Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 55–68.

[8] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Proving Differential Privacy via Probabilistic Couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '16)*. ACM, New York, NY, USA, 749–758.

[9] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic Relational Reasoning for Differential Privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 97–110.

[10] Gilles Barthe and Federico Olmedo. 2013. Beyond Differential Privacy: Composition Theorems and Relational Logic for f-divergences Between Probabilistic Programs. In *Proceedings of the 40th International Conference on Automata, Languages, and Programming - Volume Part II (ICALP'13)*. Springer-Verlag, Berlin, Heidelberg, 49–60.

[11] Dirk Beyer and M. Erkan Keremoglu. 2011. CPACHECKER: A Tool for Configurable Software Verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 184–190.

[12] Benjamin Bichsel, Timon Gehr, Dana Drachsler-Cohen, Petar Tsankov, and Martin Vechev. 2018. DP-Finder: Finding Differential Privacy Violations by Sampling and Optimization. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 508–524.

[13] Mark Bun and Thomas Steinke. 2016. Concentrated Differential Privacy: Simplifications, Extensions, and Lower Bounds. In *Proceedings, Part I, of the 14th International Conference on Theory of Cryptography - Volume 9985*. Springer-Verlag New York, Inc., New York, NY, USA, 635–658.

[14] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. 2012. SMTInterpol: An Interpolating SMT Solver. In *Proceedings of the 19th International Conference on Model Checking Software (SPIN'12)*. Springer-Verlag, Berlin, Heidelberg, 248–254.

[15] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT Solver. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*. Springer-Verlag, Berlin, Heidelberg, 93–107.

[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.

[17] Zeyu Ding, Yuxin Wang, Guanhong Wang, Danfeng Zhang, and Daniel Kifer. 2018. Detecting Violations of Differential Privacy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 475–489.

[18] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our Data, Ourselves: Privacy via Distributed Noise Generation. In *Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques (EUROCRYPT'06)*. Springer-Verlag, Berlin, Heidelberg, 486–503.

[19] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography*, Shai Halevi and Tal Rabin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 265–284.

[20] Cynthia Dwork, Aaron Roth, et al. 2014. The algorithmic foundations of differential privacy. *Theoretical Computer Science* 9, 3–4 (2014), 211–407.

[21] Hamid Ebadi, David Sands, and Gerardo Schneider. 2015. Differential Privacy: Now It's Getting Personal. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 69–81.

[22] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA, 1054–1067.

[23] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. 2017. Relational Symbolic Execution. *arXiv e-prints*, Article arXiv:1711.08349 (Nov 2017).

[24] Anna C. Gilbert and Audra McMillan. 2018. Property Testing For Differential Privacy. *2018 56th Annual Allerton Conference on Communication, Control, and Computing (Allerton)* (2018), 249–258.

[25] Moritz Hardt, Katrina Ligett, and Frank McSherry. 2012. A Simple and Practical Algorithm for Differentially Private Data Release. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2 (NIPS'12)*. Curran Associates Inc., USA, 2339–2347.

[26] Sebastian Hunt and David Sands. 2006. On Flow-sensitive Security Types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, NY, USA, 79–90.

[27] Noah Johnson, Joseph P Near, and Dawn Song. 2018. Towards practical differential privacy for SQL queries. *Proceedings of the VLDB Endowment* 11, 5 (2018), 526–539.

[28] Dexter Kozen. 1981. Semantics of probabilistic programs. *J. Comput. System Sci.* 22, 3 (1981), 328 – 350.

[29] Min Lyu, Dong Su, and Ninghui Li. 2017. Understanding the sparse vector technique for differential privacy. *Proceedings of the VLDB Endowment* 10, 6 (2017), 637–648.

[30] Frank McSherry and Kunal Talwar. 2007. Mechanism Design via Differential Privacy. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS '07)*. IEEE Computer Society, Washington, DC, USA, 94–103.

[31] Frank D. McSherry. 2009. Privacy Integrated Queries: An Extensible Platform for Privacy-preserving Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data (SIGMOD '09)*. ACM, New York, NY, USA, 19–30.

[32] I. Mironov. 2017. Rényi Differential Privacy. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. 263–275.

[33] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. 2012. GUPT: Privacy Preserving Data Analysis Made Easy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 349–360.

[34] Kobbi Nissim, Thomas Steinke, Alexandra Wood, Micah Altman, Aaron Bembenek, Mark Bun, Marco Gaboardi, David R O'Brien, and Salil Vadhan. 2017. Differential privacy: A primer for a non-technical audience. In *Privacy Law Scholars Conf.*

[35] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. 2010. Airavat: Security and Privacy for MapReduce. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*. USENIX Association, Berkeley, CA, USA, 20–20.

[36] Andrei Sabelfeld and Andrew C Myers. 2003. Language-based information-flow security. *IEEE Journal on selected areas in communications* 21, 1 (2003), 5–19.

[37] Apple Differential Privacy Team. 2017. Learning with Privacy at Scale. *Apple Machine Learning Journal* 1, 8 (2017).

[38] Tachio Terauchi and Alex Aiken. 2005. Secure information flow as a safety problem. In *International Static Analysis Symposium*. Springer, 352–367.

[39] Michael Carl Tschantz, Dilsun Kaynar, and Anupam Datta. 2011. Formal Verification of Differential Privacy for Interactive Systems (Extended Abstract). *Electronic Notes in Theoretical Computer Science* 276 (Sept. 2011), 61–79.

[40] Yuxin Wang, Zeyu Ding, Guanhong Wang, Daniel Kifer, and Danfeng Zhang. 2019. Proving Differential Privacy with Shadow Execution. *arXiv e-prints*, Article arXiv:1903.12254 (Mar 2019).

[41] Lili Xu, Konstantinos Chatzikokolakis, and Huimin Lin. 2014. Metrics for Differential Privacy in Concurrent Systems. In *Formal Techniques for Distributed Objects, Components, and Systems*, Erika Ábrahám and Catuscia Palamidessi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 199–215.

[42] Danfeng Zhang and Daniel Kifer. 2017. LightDP: Towards Automating Differential Privacy Proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 888–901.

[43] Dan Zhang, Ryan McKenna, Ios Kotsogiannis, Michael Hay, Ashwin Machanavajjhala, and Gerome Miklau. 2018. EKTELO: A Framework for Defining Differentially-Private Computations. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 115–130.