# 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


## 1 Introduction

Differential privacy is increasingly being used in industry [23, 28, 38] 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 [30], 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., [32, 34, 36, 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 [21, 30] and Report Noisy Max [21]. 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 [21]. 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, 18, 24, 25]. 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 $\epsilon$-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 [21, 30]. 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, 23, 28, 38]. There are currently several popular variants of differential privacy [13, 19, 20, 33]. In this paper, we focus on the verification of algorithms that satisfy pure differential privacy [20], 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 [35].

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.

[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 [29].

**Definition 1** (Pure Differential privacy). *Let $\epsilon \geq 0$. A probabilistic computation $M : A \to B$ is $\epsilon$-differentially private with respect to an adjacency relation $\Psi$ 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* [31].

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 $\eta$ 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 $\eta$ 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 $e^{\epsilon}$ factor results from the fact that $p_{1/\epsilon}(\eta + c) \big/ p_{1/\epsilon}(\eta) \leq e^{|c| \cdot \epsilon} \leq e^{\epsilon}$, when the Laplace distribution has scale $1/\epsilon$. The second to last equality is by change of variable from $f(\eta)$ to $\eta$ 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. $M(a_2)$ 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 [21], 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 [26].

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

```
function NoisyMax (ϵ, size : num⟨0,0⟩ ; q : list num⟨∗,∗⟩ )
        returns max : num⟨0,∗⟩
precondition ∀i ≥ 0. −1 ≤ q̂°[i] ≤ 1 ∧ q̂†[i] = q̂°[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  bq̂° := 0; bq̂† := 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 + bq̂† ∨ i = 0);
9        max := i;
10       bq̂† := bq + bq̂† - (q[i] + η);
11       bq := q[i] + η;
12       bq̂° := q̂°[i] + 2;
13    else
14       assert (¬(q[i] + q̂°[i] + η + 0 > bq + bq̂° ∨ i = 0));
15    // shadow execution
16    if (q[i] + q̂†[i] + η > bq + bq̂† ∨ i = 0)
17       bq̂† := 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 $\Omega$ 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.

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 [21]. 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.

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, based on the same argument in Section 2.2, we can prove that the Report Noisy Max algorithm is $\epsilon$-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*.

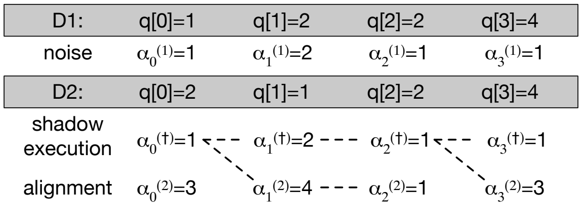

**Figure 2.** Selective alignment for Report Noisy Max

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 $\epsilon$), then a program is $\epsilon$-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 $\epsilon$ in the transformed program, then the original program being verified is $\epsilon$-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] *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.

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

**Figure 3.** ShadowDP: language syntax.

### 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 $\eta \in RVars$ to store a sample drawn from a distribution. Random variables are similar to normal variables ($x \in NVars$) 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, $\mathtt{Lap}\ r$. Semantically, $\eta := \mathtt{Lap}\ r$ draws one sample from the Laplace distribution, with mean zero and a scale factor $r$, and assigns it to $\eta$. 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 $\mathtt{num}$ (numeric type), $\mathtt{bool}$ (Boolean), or a list of elements with type $\tau$ ($\mathtt{list}\ \tau$).

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 $\mathtt{bool}$, $\mathtt{list}\ \tau$ 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, ($\mathtt{list}\ \mathtt{num}_{\langle 1,1 \rangle}$) 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 $\mathtt{num}_{\langle *,* \rangle}$ is desugared as $x : \Sigma_{(\langle \widehat{x}^\circ : \mathtt{num}_{\langle 0,0 \rangle}, \widehat{x}^\dagger : \mathtt{num}_{\langle 0,0 \rangle} \rangle)}\ \mathtt{num}_{\langle \widehat{x}^\circ, \widehat{x}^\dagger \rangle}$, where $\widehat{x}^\circ$, $\widehat{x}^\dagger$ are distinguished variables invisible in the source code; hiding those variables in a $\Sigma$-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 $\mathtt{list}\ \mathtt{num}_{\langle *,* \rangle}$, along with the precondition $\forall i \geq 0.\ -1 \leq \widehat{\mathtt{q}}^\circ\mathtt{[i]} \leq 1$. 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 ($\eta := \mathtt{Lap}\ r, \mathcal{S}, \mathbb{n}_\eta$). 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 $\eta$, 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 ($\eta := \mathtt{Lap}\ (2/\epsilon), \Omega\ ?\ \dagger : \circ, \Omega\ ?\ 2 : 0$), where $\Omega$ is $\mathtt{q[i]}+\eta > \mathtt{bq} \lor \mathtt{i} = \mathtt{0}$, 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, $[\![g]\!] : \mathbf{Dist}([\![\mathsf{num}]\!])$. Moreover, a command $c$ is interpreted as a function $[\![c]\!] : \mathcal{M} \to \mathbf{Dist}(\mathcal{M})$. For brevity, we write $[\![e]\!]_m$ and $[\![c]\!]_m$ instead of $[\![e]\!](m)$ and $[\![c]\!](m)$ hereafter. Finally, all programs have the form ($c$; `return` $e$) where $c$ contains no return statement. A program is interpreted as a function $m \to \mathbf{Dist}[\![\mathcal{B}]\!]$ where $\mathcal{B}$ is the return type (of $e$).

The semantics of commands is available in the Appendix; the semantics directly follows a standard semantics in [29].

# 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 $\Gamma$ 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 $\Psi$, 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 $\tau$ under the environment $\Gamma$. 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 [17]), 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 $\Gamma$, the new typing environment is $\Gamma'$ 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 $\Gamma$ instead of $\Gamma^\circ$ 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 $\Gamma$, then the final memories are aligned by $\Gamma'$.

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 $\Gamma_1$ and $\Gamma_2$ 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, $\eta$ has (aligned) distance $\Omega \ ?\ 2 : 0$, where $\Omega$ 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 $\Gamma_f$ 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 [27]):

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

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

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, $\Gamma''_n$ 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

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

**Typing rules for expressions**

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

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

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

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

**Typing rules for commands**

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

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

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

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

**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**

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

**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. $\Psi$ 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.

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 $\Gamma_1$ and $\Gamma_2$ 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 Appendix.

### 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 $\perp$ and $\top$ (introducing program counters in a type system is common in information flow control to track implicit flows [37]). Here, $\top$ (resp. $\perp$) means that the shadow execution might take a different branch (resp. must take the same branch) as the original execution.

When $pc = \perp$, 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* $\perp$ *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 $pc = \perp \wedge pc' = \top$. 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, 39]. 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 Appendix. 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 $\eta$, 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 $\eta$ 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 $\eta$ is generated in an injective way: the same aligned value of $\eta$ implies the same value of $\eta$ 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):

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

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

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

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

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 $\eta$ 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 $\epsilon$-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 $\epsilon$-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 $\eta$, which semantically sets the variable $\eta$ 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 $\eta$ 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 $\dagger$): 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 $\epsilon$ (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 $\epsilon$ in all possible executions of the program, then the original program satisfies $\epsilon$-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 $\epsilon$-differentially private.

## 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 $\epsilon$-differential privacy (Theorem 2): if the value of $\mathbf{v}_\epsilon$ in $c''$ is always bounded by a constant $\epsilon$, then $c$ is $\epsilon$-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 Appendix.

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

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

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.

**$\Gamma$-*Relation*** To formalize and prove the soundness property, we notice that a typing environment $\Gamma$ along with distance environment $\gamma$ 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 $\Gamma^{\circ}$ (resp. $\Gamma^{\dagger}$) and $\gamma$. Intuitively, the initial $\gamma$ and $\Gamma$ (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 $\gamma$ and $\Gamma$ in Figure 1 specifies that two executions of the program is related if non-private variables $\epsilon$, *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** ($\Gamma$-Relations). *Two memories $m_1'$ (in the form of $m_1 \uplus (\gamma)$) and $m_2$ are related by $\Gamma^\circ$, written $m_1' \;\Gamma^\circ\; m_2$, if $\forall x \in \mathrm{Vars} \cup \mathrm{RVars}$, 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, $\Gamma^\circ$ 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 $\gamma$, both $\Gamma^\circ$ and $\Gamma^\dagger$ must be injective functions [42]. The injectivity of $\Gamma$ over the entire memory follows from the injectivity of $\Gamma$ over the random noises $\eta \in \mathit{RVars}$, 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 $\eta \in \mathit{RVars}$.

**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 $\epsilon$, there is an extension of $m'$, written $m' \uplus (\epsilon)$, so that*

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

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 $c'' \upharpoonright^{m_2'}_{m_1'}$, is defined as*

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

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

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

**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:*

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

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

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 $\epsilon$, 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 $\epsilon$.

**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 $\max(c'' \upharpoonright^{m_2'}_{m_1'}) \leq \epsilon$ 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

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

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 $\Gamma_1^\circ = \Gamma_1^\dagger \wedge \bot \vdash \Gamma_1\{(c; \mathtt{return}\ e) \rightharpoonup (c'; \mathtt{return}\ e)\}\Gamma_2 \;\wedge\; c' \rightrightarrows c''$, we have*

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

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

By rule (T-Return), $\Gamma_2 \vdash e : \mathtt{num}_{\langle 0, \mathbb{d}\rangle}$ or $\Gamma_2 \vdash e : \mathtt{bool}$. 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 $\Gamma_2 \vdash e : \mathtt{num}_{\langle 0, \mathbb{n}\rangle}$, and rule (T-ODot) when $\Gamma_2 \vdash e : \mathtt{bool}$. We note that $\Gamma_2^\circ S_V' \neq S_V$ in general since $\Gamma_2^\circ$ might not be a surjection. Let $P' = (c'; \mathtt{return}\ e)$, then for any $\gamma$, we have

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

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

***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` ($\mathbf{v}_\epsilon \le \epsilon$) 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` ($\mathbf{v}_\epsilon \le \epsilon$) 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 [16], Z3 [17] and SMTInterpol [15]. 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 $\epsilon$ 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 $\epsilon$, 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 Appendix.

### 6.2.1 Sparse Vector Technique

Sparse Vector Technique [21] is a powerful mechanism which has been proven to satisfy $\epsilon$-differential privacy (its proof is notoriously tricky to write manually [30]). 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 [21]. 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 [21]. As shown in Figure 6, programmers only have to provide two simple annotations: $\circ$ , 1 for $\eta_1$ and $\circ$ , $\Omega$ ? 2 : 0 for $\eta_2$. Since the selectors in this example only 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 $\epsilon$-differential privacy (we also verified a variant where $\epsilon$ 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 $\mathtt{q[i]} + \eta_2 \ge \tilde{T}$, it outputs the value of the gap $\mathtt{q[i]} + \eta_2 - \tilde{T}$ (how much larger the noisy answer is

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

**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 $\epsilon$** | |
| 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⟨0,0⟩ ; q : list num⟨*,*⟩ )
    returns (out : list bool )
precondition ∀i ≥ 0. −1 ≤ q̂°[i] ≤ 1 ∧ q̂†[i] = q̂°[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   vε := 0;
2   havoc η1; vε := vε + 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; vε = Ω ? (vε + 2 × ε/4N) : (vε + 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 $\Omega$ represents the branch condition.

compared to the noisy threshold). Note that the noisy query value $\mathtt{q[i]} + \eta_2$ is reused for both this check and the output (whereas other proposals either (1) draw fresh noise and result in a larger $\epsilon$ [21], or (2) re-use the noise but do not satisfy differential privacy, as noted in [30]). 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 $\eta_2$ from $\Omega\ ?\ 2 : 0$ to $\Omega\ ?\ (1 - \widehat{\mathtt{q}}^{\circ}[\mathtt{i}]) : 0$, 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 $\mathbf{v}_\epsilon := \mathbf{v}_\epsilon + (1 - \widehat{\mathtt{q}}^{\circ}[\mathtt{i}]) \times \epsilon/4N;$ to **assert** $(|1 - \widehat{\mathtt{q}}^{\circ}[\mathtt{i}]| \leq 2)$; $\mathbf{v}_\epsilon := \mathbf{v}_\epsilon + 2 \times \epsilon/4N;$ and provide nonlinear loop invariants; then it is verified (we also verified a variant where $\epsilon$ 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 Appendix. 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 $\epsilon$-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 $\Omega$, 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) [22] allows each individual to set its own different privacy level and PDP will satisfy difference privacy regarding the level she sets. PINQ [32] 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 [40, 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] T.-H. Hubert Chan, Elaine Shi, and Dawn Song. 2011. Private and Continual Release of Statistics. *ACM Trans. Inf. Syst. Secur.* 14, 3, Article 26 (Nov. 2011), 24 pages.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

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

[34] 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.

[35] 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.*

[36] 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.

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

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

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

[40] 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.

[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.

# Appendix

## A ShadowDP Semantics

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

**Figure 7.** ShadowDP: language semantics.

Given a distribution $\mu \in \mathbf{Dist}(A)$, its support is defined as $\mathtt{support}(\mu) \triangleq \{a \mid \mu(a) > 0\}$. We use $\mathbb{1}_a$ to represent the degenerate distribution $\mu$ that $\mu(a) = 1$ and $\mu(a') = 0$ if $a' \neq a$. Moreover, we define monadic functions unit and bind functions to formalize the semantics for commands:

$$
\begin{aligned}
&\mathtt{unit} : A \to \mathbf{Dist}(A) \triangleq \lambda a.\ \mathbb{1}_a \\
&\mathtt{bind} : \mathbf{Dist}(A) \to (A \to \mathbf{Dist}(B)) \to \mathbf{Dist}(B) \\
&\quad \triangleq \lambda\mu.\ \lambda f.\ (\lambda b.\ \sum_{a \in A} (f\ a\ b) \times \mu(a))
\end{aligned}
$$

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 $B$ (e.g., a conditional distribution of $B$ given $A$), and returns the corresponding marginal distribution on $B$. This monadic view avoids cluttered definitions and proofs when probabilistic programs are involved.

Figure 7 provides the semantics of commands.

## B Constructing Shadow Execution

Figure 8 shows the rule for generating shadow execution expressions as well as aligned execution expressions. Figure 9 shows the rules for generating shadow execution commands. The shadow execution essentially replaces each variables $x$ with their correspondence (i.e., $x + \Gamma(x)$) in $c$, as standard in self-composition construction [4, 39]. Compared with standard self-composition, the differences are:

1. $(\!|c, \Gamma|\!)^\dagger$ is not applicable to sampling commands, since if the original execution takes a sample while the shadow execution does not, we are unable to align the sample variable due to different probabilities.
2. For convenience, we use $x + \mathbb{n}^\dagger$ where $\Gamma \vdash x : \langle \mathbb{n}^\circ, \mathbb{n}^\dagger \rangle$ whenever the shadow value of $x$ is used; correspondingly, we update $\widehat{x}^\dagger$ to $v - x$ instead of updating the shadow value of $x$ to some value $v$.

$$
(\!|r, \Gamma|\!)^\star = r \quad (\!|\mathtt{true}, \Gamma|\!)^\star = \mathtt{true} \quad (\!|\mathtt{false}, \Gamma|\!)^\star = \mathtt{false}
$$

$$
(\!|x, \Gamma|\!)^\star = \begin{cases} x + \mathbb{n}^\star & \text{, else if } \Gamma \vdash x : \mathsf{num}_{\langle \mathbb{n}^\circ, \mathbb{n}^\dagger \rangle} \\ x & \text{, else} \end{cases}
$$

$$
(\!|e_1 \ \mathsf{op}\ e_2, \Gamma|\!)^\star = (\!|e_1, \Gamma|\!)^\star \ \mathsf{op}\ (\!|e_2, \Gamma|\!)^\star \text{ where } \mathsf{op} = \oplus \cup \otimes \cup \odot
$$

$$
(\!|e_1[e_2], \Gamma|\!)^\star = \begin{cases} e_1[e_2] + \widehat{e_1}^\star[e_2] & \text{, if } \Gamma^\star \vdash e_1 : \mathtt{list}\ \mathsf{num}_* \\ e_1[e_2] + \mathbb{n}^\star & \text{, else if } \Gamma^\star \vdash e_1 : \mathtt{list}\ \mathsf{num}_{\mathbb{n}^\star} \\ e_1[e_2] & \text{, else} \end{cases}
$$

$$
(\!|e_1 :: e_2, \Gamma|\!)^\star = (\!|e_1, \Gamma|\!)^\star :: (\!|e_2, \Gamma|\!)^\star \qquad (\!|\neg e, \Gamma|\!)^\star = \neg(\!|e, \Gamma|\!)^\star
$$

$$
(\!|e_1\ ?\ e_2 : e_3, \Gamma|\!)^\star = (\!|e_1|\!)^\star\ ?\ (\!|e_2, \Gamma|\!)^\star : (\!|e_3, \Gamma|\!)^\star
$$

**Figure 8.** Transformation of numerical expressions for aligned and shadow execution, where $\star \in \{\circ, \dagger\}$.

$$
(\!|\mathtt{skip}, \Gamma|\!)^\dagger = \mathtt{skip} \qquad \frac{(\!|c_1; \Gamma|\!)^\dagger = c_1' \quad (\!|c_2; \Gamma|\!)^\dagger = c_2'}{(\!|c_1; c_2, \Gamma|\!)^\dagger = c_1'; c_2'}
$$

$$
(\!|x := e, \Gamma|\!)^\dagger = (\widehat{x}^\dagger := (\!|e, \Gamma|\!)^\dagger - x)
$$

$$
\frac{(\!|c_i, \Gamma|\!)^\dagger = c_i' \quad i \in \{1, 2\}}{(\!|\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \Gamma|\!)^\dagger = \mathtt{if}\ (\!|e, \Gamma|\!)^\dagger\ \mathtt{then}\ c_1'\ \mathtt{else}\ c_2'}
$$

$$
\frac{(\!|c, \Gamma|\!)^\dagger = c'}{(\!|\mathtt{while}\ e\ \mathtt{do}\ c, \Gamma|\!)^\dagger = \mathtt{while}\ (\!|e, \Gamma|\!)^\dagger\ \mathtt{do}\ c'}
$$

**Figure 9.** Shadow execution for commands.

## C Extra Case Studies

In this section, we study extra differentially private algorithms to show the power of ShadowDP. As Section 6.2, the shadow execution part is optimized away when the selectors never use the shadow variables.

### C.1 Numerical Sparse Vector Technique

Numerical Sparse Vector Technique [21] is an interesting variant of Sparse Vector Technique which outputs numerical query answers when the query answer is large. To achieve differential privacy, like Sparse Vector Technique, it adds noise to the threshold $T$ and each query answer q[i]; it then tests if the noisy query answer is above the noisy threshold or not. The difference is that Numerical Sparse Vector Technique draws a fresh noise $\eta_3$ when the noisy query answer is above the noisy threshold, and then releases q[i]$+\eta_3$ instead of simply releasing `true`. The pseudo code for this algorithm is shown in Figure 10.

In this algorithm, ShadowDP needs an extra annotation for the new sampling command of $\eta_3$. We use the same approach in Gap Sparse Vector Technique for this new sampling command. Recall the observation that we want the final output variable `out` to have distance $\langle 0, -\rangle$, which implies that the numerical query $\texttt{q[i]}+\eta_3$ should have distance $\langle 0, -\rangle$. We can deduce that $\eta_3$ must have distance $-\widehat{\texttt{q}}^{\circ}\texttt{[i]}$ inside the branch, thus we write "$\circ, -\widehat{\texttt{q}}^{\circ}\texttt{[i]}$" for $\eta_3$. The rest of the annotations remain the same as standard Sparse Vector Technique.

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

```
η1 := Lap (3/ϵ) , ◦, 1 ;
T̃ := T + η1; count := 0; i := 0;
while (count < N ∧ i < size)
  η2 := Lap (6N/ϵ) , ◦, Ω ? 2 : 0 ;
  if (q[i] + η2 ≥ T̃) then
    η3 := Lap (3N/ϵ) , ◦, −q̂°[i] ;
    out := (q[i] + η3)::out;
    count := count + 1;
  else
    out := 0::out;
  i := i + 1;
```

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

```
vϵ := 0;
havoc η1; vϵ := vϵ + ϵ/3;
T̃ := T + η1;
count := 0; i := 0;
while (count < N ∧ i < size)
  assert (count < N ∧ i < size);
  havoc η2; vϵ = Ω ? (vϵ + 2 × ϵ/6N) : (vϵ + 0);
  if (q[i] + η2 ≥ T̃) then
    assert (q[i] + q̂°[i] + η2 + 2 ≥ T̃ + 1);
    havoc η3; vϵ = vϵ + | − q̂°[i]| × ϵ/3N;
    out := (q[i] + η3)::out;
    count := count + 1;
  else
    assert (¬(q[i] + q̂°[i] + η2 ≥ T̃ + 1));
    out := 0::out;
  i := i + 1;
```

**Figure 10.** Verifying Numerical Sparse Vector Technique with ShadowDP. Annotations are in gray where $\Omega$ represents the branch condition.

For the non-linear issue of the verifier, we rewrite the privacy cost assignment from

$$\mathbf{v}_\epsilon = \mathbf{v}_\epsilon + |-\widehat{\texttt{q}}^{\circ}[\texttt{i}]| \times \epsilon/3;$$

$$\text{to}\quad \textbf{assert}\ (|-\widehat{\texttt{q}}^{\circ}[\texttt{i}]| \leq 1);\ \mathbf{v}_\epsilon = \mathbf{v}_\epsilon + \epsilon/3;$$

With manual loop invariants provided, CPAChecker successfully verified the rewritten program.

### C.2 Partial Sum

We now study an $\epsilon$-differentially private algorithm PartialSum (Figure 11) which simply sums over a list of queries. To achieve differential privacy, it adds noise using Laplace mechanism to the final sum and output the noisy sum. One difference from the examples in Section 6.2 is the adjacency assumption: at most one query answer may differ by 1 as specified in the precondition.

In this example, since the noise is added to the final `sum`, it makes no difference if we choose the aligned version or shadow version of normal variables (they are both identical to the original execution). To decide the distance of the random variable, we want the final output to have distance $\langle 0, -\rangle$, it is easy to deduce that the distance of $\eta$ should be $-\widehat{\texttt{sum}}^{\circ}$. Adding the annotation $\circ,\ -\widehat{\texttt{sum}}^{\circ}$ to Line 5 in Figure 11, ShadowDP successfully transforms and verifies the program. Note that since the cost update command $\mathbf{v}_\epsilon := \mathbf{v}_\epsilon + |\widehat{\texttt{sum}}| \times \epsilon$; contains non-linear arithmetic, we carefully rewrite this command to `assert` ($|\widehat{\texttt{sum}}| \leq 1$); $\mathbf{v}_\epsilon := \mathbf{v}_\epsilon + \epsilon$;. ShadowDP is able to type check and verify this algorithm within seconds.

**function** PartialSum ($\epsilon$, size $: \mathsf{num}_{\langle 0,0\rangle}$ ; q $: \mathsf{list}\ \mathsf{num}_{\langle *,*\rangle}$ )
**returns** (out $: \mathsf{num}_{\langle 0,-\rangle}$ )
**precondition** $\forall i \geq 0.\ -1 \leq \widehat{\texttt{q}}^{\circ}[\texttt{i}] \leq 1 \ \wedge\ \widehat{\texttt{q}}^{\dagger}[\texttt{i}] = \widehat{\texttt{q}}^{\circ}[\texttt{i}] \ \wedge$
$\widehat{\texttt{q}}^{\circ}[\texttt{i}] \neq 0 \Rightarrow (\forall j > i.\ \widehat{\texttt{q}}^{\circ}[\texttt{j}] = 0)$

```
1 sum := 0; i := 0;
2 while (i < size)
3   sum := sum + q[i];
4   i := i + 1;
5 η = Lap (1/ϵ) , ◦, − sum^° ;
6 out := sum + η;
```

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

```
sum := 0; i := 0;
sum^° := 0;
while (i < size)
  assert (i < size);
  sum := sum + q[i];
  sum^° := sum^° + q̂°[i];
  i := i + 1;
havoc η; vϵ := vϵ + |sum^°| × ϵ;
out := sum + η;
```

**Figure 11.** Verifying Partial Sum using ShadowDP. Annotations are shown in gray.

### C.3 Smart Sum and Prefix Sum

Another interesting algorithm SmartSum [14] has been previously verified [6, 9] with heavy annotations. We illustrate the power of our type system by showing that this algorithm can be verified with very little annotation burden for the programmers.

```
function SmartSum (ε, M, T : num⟨0,0⟩  q : list num⟨*,*⟩ )
         returns (out : list num⟨0,−⟩ )
precondition ∀i ≥ 0. −1 ≤ q̂°[i] ≤ 1 ∧ q̂†[i] = q̂°[i] ∧
                     q̂°[i] ≠ 0 ⇒ (∀j > i. q̂°[j] = 0)

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

The transformed program:

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

**Figure 12.** Verifying SmartSum algorithm with ShadowDP. Annotations are shown in gray.

This algorithm (Figure 12) is designed to continually release aggregate statistics in a privacy-preserving manner. The annotations are only needed on Line 4 and 9 in Figure 12. We use the same observation as stated in Section 6.2.2, in order to make the aligned distance of the output variable out 0. To do that, we assign distance $-\widehat{\texttt{sum}}^\circ - \widehat{\texttt{q}}^\circ\texttt{[i]}$ to $\eta_1$ and $-\widehat{\texttt{q}}^\circ\texttt{[i]}$ to $\eta_2$, and use $\circ$ for both random variables. This algorithm is successfully transformed by ShadowDP. However, due to the non-linear issue described in Section 6.1, we change the commands of Line 6 and Line 12 from

$$\mathbf{v}_\epsilon := \mathbf{v}_\epsilon + |-\widehat{\texttt{sum}}^\circ - \widehat{\texttt{q}}^\circ\texttt{[i]}| \times \epsilon;$$
$$\mathbf{v}_\epsilon := \mathbf{v}_\epsilon + |-\widehat{\texttt{q}}^\circ\texttt{[i]}| \times \epsilon;$$

to

```
if (|−sum^° − q̂°[i]| > 0)
   assert (|−sum^° − q̂°[i]| ≤ 1); v_ε := v_ε + ε;
if (|−q̂°[i]| > 0)
   assert (|−q̂°[i]| ≤ 1); v_ε := v_ε + ε;
```

Moreover, one difference of this algorithm is that it satisfies $2\epsilon$-differential privacy [14] instead of $\epsilon$-differential privacy, thus the last assertion added to the program is changed to assert $(\mathbf{v}_\epsilon \le 2 \times \epsilon)$;. With this CPAChecker is able to verify this algorithm.

We also verified a variant of Smart Sum, called Prefix Sum algorithm [2]. This algorithm is a simplified and less precise version of Smart Sum, where the else branch is always taken. More specifically, we can get Prefix Sum by removing Lines 3 - 8 from Figure 12. The annotation remains the same for $\eta_2$, type checking and transformation follows Smart Sum. Note that Prefix Sum satisfies $\epsilon$-differential privacy, so the last assertion remains unchanged. CPAChecker then verifies the transformed Prefix Sum within 2 seconds.

# D Soundness Proof

We first prove a couple of useful lemmas. Following the same notations as in Section 5, we use $m$ for original memory and $m'$ for extended memory with aligned and shadow variables but not the distinguished privacy tracking variable $\mathbf{v}_\epsilon$. Moreover, we assume that memory tracks the entire list of sampled random values at each point.

**Lemma 4** (Numerical Expression). *$\forall e, m', \Gamma$, such that $\Gamma \vdash e : \texttt{num}_{\langle \mathbb{n}^\circ, \mathbb{n}^\dagger \rangle}$, we have*

$$[\![e]\!]_{m'} + [\![\mathbb{n}^\circ]\!]_{m'} = [\![e]\!]_{\Gamma^\circ m'},$$
$$[\![e]\!]_{m'} + [\![\mathbb{n}^\dagger]\!]_{m'} = [\![e]\!]_{\Gamma^\dagger m'}.$$

*Proof.* Induction on the inference rules.

- (T-Num): trivial.
- (T-Var): If $\Gamma^\circ(x)$ is $*$, $\mathbb{n}^\circ$ is $\widehat{x}^\circ$. By Definition 3, $\Gamma^\circ m'(x) = m'(x) + m'(\widehat{x}^\circ)$. If $\Gamma^\circ(x) = \mathbb{d}^\circ$, $\mathbb{n}^\circ$ is $\mathbb{d}^\circ$. Again, we have $\Gamma^\circ m'(x) = m'(x) + [\![\mathbb{d}^\circ]\!]_{m'}$ by Definition 3. The case for $\Gamma^\dagger$ is similar.
- (T-OPlus, T-OTimes, T-Ternary, T-Index): by the induction hypothesis (list is treated as a collection of variables of the same type).

□

**Lemma 5** (Boolean Expression). *$\forall e, m', \Gamma$, such that $\Gamma \vdash e : \texttt{bool}$, we have*

$$[\![e]\!]_{m'} = [\![e]\!]_{\Gamma^\circ m'} = [\![e]\!]_{\Gamma^\dagger m'}.$$

*Proof.* Induction on the inference rules.

- (T-Boolean): trivial.
- (T-Var): the special case where $\mathbb{d}^\circ = \mathbb{d}^\dagger = 0$. Result is true by Definition 3.
- (T-ODot): by Lemma 4 and induction hypothesis, we have $[\![e_i]\!]_{m'} + [\![\mathbb{n}_i^\circ]\!]_{m'} = [\![e_i]\!]_{\Gamma^\circ m'}$ for $i \in \{1, 2\}$. By the assumption of (T-ODot), we have

$$[\![e_1 \odot e_2]\!]_{m'} \Leftrightarrow [\![(e_1 + \mathbb{n}_1^\circ) \odot (e_2 + \mathbb{n}_2^\circ)]\!]_{m'} = [\![e_1 \odot e_2]\!]_{\Gamma^\circ m'}$$

  The case for $\Gamma^\dagger$ is similar.
- (T-Ternary, T-Index): by the induction hypothesis.

□

## D.1 Injectivity

We first prove that the type system maintains injectivity.

**Lemma 6.** $\forall c, c', pc, m', m_1', m_2', \Gamma_1, \Gamma_2.\ pc \vdash \Gamma_1\{c \rightharpoonup c'\}\Gamma_2 \wedge [\![c']\!]_{m'} m_1' \neq 0 \wedge [\![c']\!]_{m'} m_2' \neq 0$, *then we have*

$$(m_1' = m_2') \vee (\exists \eta.\ \Gamma_2^\star m_1'(\eta) \neq \Gamma_2^\star m_2'(\eta)), \star \in \{\circ, \dagger\}$$

*Proof.* By structural induction on $c$.

- Case `skip`: trivial since it is non-probabilistic.
- Case $x := e$: trivial since it is non-probabilistic.
- Case $c_1; c_2$: Let $pc \vdash \Gamma_1\{c_1 \rightharpoonup c_1'\}\Gamma$, $pc \vdash \Gamma\{c_2 \rightharpoonup c_2'\}\Gamma_2$. There exists some $m_3'$ and $m_4'$ such that

$$[\![c_1']\!]_{m'}(m_3') \neq 0 \wedge [\![c_2']\!]_{m_3'}(m_1') \neq 0$$
$$[\![c_1']\!]_{m'}(m_4') \neq 0 \wedge [\![c_2']\!]_{m_4'}(m_2') \neq 0$$

  If $m_3' = m_4'$, results is true by the induction hypothesis on $c_2'$. Otherwise, we have $\exists \eta.\ \Gamma_2^\star m_3'(\eta) \neq \Gamma_2^\star m_4'(\eta)$. Hence, $\Gamma_2^\star m_1'(\eta) \neq \Gamma_2^\star m_2'(\eta)$; contradiction.
- Case `if` $e$ `then` $c_1$ `else` $c_2$: let $pc' \vdash \Gamma_1\{c_i \rightharpoonup c_i'\}\Gamma_i'$. By typing rule, we have $\Gamma_1' \sqcup \Gamma_2' = \Gamma_2$. Given initial memory $m'$, both executions take the same branch. Without loss of generality assume $c_1$ is executed. Let $[\![c_1']\!]_{m'} m_3' \neq 0$, $[\![c_1'']\!]_{m_3'} m_1' \neq 0$, $[\![c_1']\!]_{m'} m_4' \neq 0$ and $[\![c_1'']\!]_{m_4'} m_1' \neq 0$. There are two cases:
  1. $m_3' = m_4'$: in this case, $c_1''$ only changes the value of $x^\star$ to $\mathbb{n}$ when $\Gamma_2(x) = *$ and $\Gamma_1'(x) = \mathbb{n}$. Moreover, we have $m_1'(x^\star) = [\![\mathbb{n}]\!]_{m_3'} = [\![\mathbb{n}]\!]_{m_4'} = m_2'(x^\star)$ for those variables. Hence, $m_1' = m_2'$.
  2. $m_3' \neq m_4'$: by induction hypothesis, $\exists \eta.\ m_3'(\eta) \neq m_4'(\eta)$. Hence, $m_1' \neq m_2'$, and $m_1'(\eta) \neq m_2'(\eta)$ for the same $\eta$.

  When $pc = \top$, $c'$ also includes $c^\dagger$. In this case, result still holds after $c^\dagger$ since $c^\dagger$ is deterministic by construction.
- Case `while` $e$ `do` $c$: let $pc' \vdash \Gamma \sqcup \Gamma_f\{c \rightharpoonup c'\}\Gamma_f$. We proceed by induction on the number of iterations:
  1. $c$ is not executed: $m_1' = m' = m_2'$.
  2. $c$ is executed $n + 1$ times: similar to the $c_1; c_2$ case.

  When $pc = \top$, $c'$ also includes $c^\dagger$. In this case, result still holds after $c^\dagger$ since $c^\dagger$ is deterministic by construction.
- Case ($\eta$ := `Lap` $r; \mathcal{S}, \mathbb{n}_\eta$): if $m_1' \neq m_2'$, we know that $m_1'(\eta) \neq m_2'(\eta)$ since $c'$ only modifies the value of $\eta$. In this case, it must be true that $\Gamma_2^\star m_1'(\eta) \neq \Gamma_2^\star m_2'(\eta)$: otherwise, $m_1' = m_2'$ by the injectivity check in rule (T-Laplace).

□

***Proof of Lemma 2***

*Proof.* Direct implication of Lemma 6. □

## D.2 Instrumentation

Next, we show a property offered by the auxiliary function $\Gamma_1, \Gamma_2, pc \Rrightarrow c'$ used in the typing rules. Intuitively, they allow us to freely switch from one typing environment to another by promoting some variables to star type.

**Lemma 7** (Instrumentation). $\forall pc, \Gamma_1, \Gamma_2, c'$, *if* $(\Gamma_1, \Gamma_2, pc) \Rrightarrow c'$, *then for any memory* $m_1'$, *there is a unique* $m_2'$ *such that* $[\![c']\!]_{m_1'}(m_2') \neq 0$ *and*

$$\begin{cases} \Gamma_1^\circ m_1' = \Gamma_2^\circ m_2' \wedge \Gamma_1^\dagger m_1' = \Gamma_2^\dagger m_2' & \text{if } pc = \bot \\ \Gamma_1^\circ m_1' = \Gamma_2^\circ m_2' & \text{if } pc = \top \end{cases}$$

*Proof.* By construction, $c'$ is deterministic. Hence, there is a unique $m_2'$ such that $[\![c]\!]_{m_1'}(m_2') \neq 0$.

Consider any variable $x \in \mathit{Vars}$. By the construction of $c'$, we note that $m_1'(x) = m_2'(x)$ and $\widehat{x}^\circ$ differs in $m_1'$ and $m_2'$ only if $\Gamma_1^\circ(x) = \mathbb{n}$ and $\Gamma_2^\circ(x) = *$. In this case, $\Gamma_1^\circ m_1'(x) = m_1'(x) + [\![\mathbb{n}]\!]_{m_1'} = m_2'(x) + m_2'(\widehat{x}^\circ) = \Gamma_2^\circ m_2'(x)$. Otherwise, $\Gamma_1^\circ(x) = \Gamma_2^\circ(x)$ (since $\Gamma_1 \sqsubseteq \Gamma_2$). When $\Gamma_1^\circ(x) = \Gamma_2^\circ(x) = *$, $\Gamma_1^\circ m_1'(x) = m_1'(x) + m_1'(\widehat{x}^\circ) = m_2'(x) + m_2'(\widehat{x}^\circ) = \Gamma_2^\circ m_2'(x)$. When $\Gamma_1^\circ(x) = \Gamma_2^\circ(x) = \mathbb{n}$ for some $\mathbb{n}$, $\Gamma_1^\circ m_1'(x) = m_1'(x) + [\![\mathbb{n}]\!]_{m_1'} = m_2'(x) + [\![\mathbb{n}]\!]_{m_2'} = \Gamma_2^\circ m_2'(x)$.

When $pc = \bot$, the same argument applies to the case of $\Gamma_1^\dagger m_1' = \Gamma_2^\dagger m_2'$. □

## D.3 Shadow Execution

Next, we show the main properties related to shadow execution.

**Lemma 8.** $\forall e, \Gamma, m'$, *if* $e$ *is well-typed under* $\Gamma$, *we have*

$$[\![(\!|e, \Gamma|\!)^\dagger]\!]_{m'} = [\![e]\!]_{\Gamma^\dagger m'}$$

*Proof.* By structural induction on the $e$.

- $e$ is $r$, `true` or `false`: trivial.
- $e$ is $x$: if $\Gamma^\dagger(x) = *$ (with base type of $\mathsf{num}$), $[\![(\!|x, \Gamma|\!)^\dagger]\!]_{m'} = m'(x) + m'(\widehat{x}^\dagger) = [\![x]\!]_{\Gamma^\dagger m'}$. When $\Gamma(x) = \mathsf{num}_{\langle \mathbb{d}, \mathbb{n}^\dagger \rangle}$, we have $[\![(\!|x, \Gamma|\!)^\dagger]\!]_{m'} = [\![x + \mathbb{n}^\dagger]\!]_{m'} = m'(x) + [\![\mathbb{n}^\dagger]\!]_{m'} = [\![x]\!]_{\Gamma^\dagger m'}$ (by Lemma 4). Otherwise, $[\![(\!|x, \Gamma|\!)^\dagger]\!]_{m'} = m'(x) = \Gamma^\dagger m'(x)$.
- $e$ is $e_1$ `op` $e_2$: by induction hypothesis.
- $e$ is $e_1[e_2]$: By the typing rule (T-Index), $\Gamma \vdash e_2 : \mathsf{num}_{\langle 0, 0 \rangle}$. Hence, $[\![e_2]\!]_{\Gamma^\dagger m'} = [\![e_2]\!]_{m'}$ by Lemma 4.
  Let $\Gamma \vdash e_1$ : `list` $\mathsf{num}_{\langle \mathbb{d}, \mathbb{d}^\dagger \rangle}$. When $e_1 = q$ and $\mathbb{d}^\dagger = *$, $[\![(\!|q[e_2], \Gamma|\!)^\dagger]\!]_{m'} = [\![q[e_2] + \widehat{q}^\dagger[e_2]]\!]_{m'} = [\![q[e_2]]\!]_{\Gamma^\dagger m'}$. When $\mathbb{d} = *$ and $e_1$ is not a variable, $[\![(\!|e_1[e_2], \Gamma|\!)^\dagger]\!]_{m'}$ is defined

as $[\![(\!|e_1,\Gamma|\!)^\dagger[e_2]]\!]_{m'}$. By induction hypothesis, this is the same as $[\![e_1]\!]_{\Gamma^\dagger m'}[\![[e_2]]\!]_{m'} = [\![e_1[e_2]]\!]_{\Gamma^\dagger m'}$.
When $\mathbb{d} = \mathbb{n}^\dagger$, $[\![(\!|e_1[e_2],\Gamma|\!)^\dagger]\!]_{m'}$ is defined as $[\![e_1[e_2] + \mathbb{n}^\dagger]\!]_{m'}$, which is the same as $[\![e_1[e_2]]\!]_{\Gamma^\dagger m'}$ (by Lemma 4).
Otherwise, result is true by Lemma 5.

- $e$ is $e_1 :: e_2$ and $\neg e$: by induction hypothesis.
- $e$ is $e_1 ? e_2 : e_3$: by induction hypothesis, we have $[\![(\!|e_1,\Gamma|\!)^\dagger]\!]_{m'} = [\![e_1]\!]_{\Gamma^\dagger m'}$. Hence, the same element is selected on both ends. Result is true by induction hypothesis.

□

Next, we show that shadow execution simulates two executions on $\Gamma^\dagger$-related memories via two program executions.

**Lemma 9** (Shadow Execution). *$\forall c, c^\dagger, \Gamma.\ (\forall x \in \mathtt{Asgnd}(c).\ \Gamma^\dagger(x) = *)\ \wedge\ (\!|c,\Gamma|\!)^\dagger = c^\dagger$, we have*

$$\forall m_1', m_2'.\ [\![c^\dagger]\!]_{m_1'}(m_2') = [\![c]\!]_{\Gamma^\dagger m_1'}(\Gamma^\dagger m_2').$$

*Proof.* By structural induction on $c$. First, we note that the construction of shadow execution does not apply to sampling instructions. Hence, $c$ and $c^\dagger$ fall into the deterministic portion of programs. Therefore, we write $m_2 = [\![c]\!]_{m_1}$ when $m_2$ is the unique memory such that $[\![c]\!]_{m_1}(m_2) = 1$, and likewise for $c'$. Then this lemma can be stated as

$$m_2' = [\![c^\dagger]\!]_{m_1'} \implies \Gamma^\dagger m_2' = [\![c]\!]_{\Gamma^\dagger m_1'}.$$

- $c$ is skip: we have $m_1' = m_2'$ in this case. Hence, $\Gamma^\dagger m_2' = \Gamma^\dagger m_1' = [\![\mathtt{skip}]\!]_{\Gamma^\dagger m_1'}$.
- $c$ is $(c_1; c_2)$: let $(\!|c_1,\Gamma|\!)^\dagger = c_1^\dagger$ and $(\!|c_2,\Gamma|\!)^\dagger = c_2^\dagger$ and $m' = [\![c_1^\dagger]\!]_{m_1'}$. By induction hypothesis, we have $\Gamma^\dagger m' = [\![c_1]\!]_{\Gamma^\dagger m_1'}$.
By language semantics, $m_2' = [\![c_2^\dagger]\!]_{m'}$. By induction hypothesis, with $\Gamma^\dagger$ as both the initial and final environments, we have $\Gamma^\dagger m_2' = [\![c_2]\!]_{\Gamma^\dagger m'} = [\![c_1; c_2]\!]_{\Gamma^\dagger m_1'}$
- $c$ is $x := e$: we have $c^\dagger = (\widehat{x}^\dagger := (\!|e,\Gamma|\!)^\dagger - x)$ in this case. Moreover, $m_2' = m_1'\{[\![(\!|e,\Gamma|\!)^\dagger) - x]\!]_{m_1'}/\widehat{x}^\dagger\}$. By Lemma 8, we have $[\![(\!|e,\Gamma|\!)^\dagger]\!]_{m_1'} = [\![e]\!]_{\Gamma^\dagger m_1'}$.
For variable $x$, we know that $\Gamma(x)^\dagger = *$ by assumption. So $\Gamma^\dagger m_2'(x) = m_2'(x) + m_2'(\widehat{x}^\dagger) = m_1'(x) + [\![(\!|e,\Gamma_2|\!)^\dagger - x]\!]_{m_1'} = [\![e]\!]_{\Gamma^\dagger m_1'} = [\![x := e]\!]_{\Gamma^\dagger m_1'}(x)$. For a variable $y$ other than $x$, the result is true since both $c$ and $c^\dagger$ do not modify $y$ and $y^\dagger$, and $y$'s distances does not change due to the well-formedness check in typing rule (T-Asgn).
- $c$ is if $e$ then $c_1$ else $c_2$: let $(\!|c_1,\Gamma|\!)^\dagger = c_1'$ and $(\!|c_2,\Gamma|\!)^\dagger = c_2'$. In this case, $c^\dagger = \mathtt{if}\ (\!|e,\Gamma|\!)^\dagger\ \mathtt{then}\ c_1'\ \mathtt{else}\ c_2'$. By Lemma 8, we have $[\![(\!|e,\Gamma|\!)^\dagger]\!]_{m_1} = [\![e]\!]_{\Gamma^\dagger m_1}$. Hence, both $c$ and $c^\dagger$ will take the same branch under $\Gamma^\dagger m_1'$ and $m_1'$ respectively. The desired results follow from induction hypothesis on $c_1$ or $c_2$.
- $c$ is while $e$ do $c$: again, since $[\![(\!|e,\Gamma|\!)^\dagger]\!]_{m_1} = [\![e]\!]_{\Gamma^\dagger m_1}$. The desired result follows from induction on the number of loop iterations.

□

Next, we show that when $pc = \top$ (i.e., when the shadow execution might diverge), the transformed code does not modify shadow variables and their distances.

**Lemma 10** (High PC). *$\forall c, c', \Gamma_1, \Gamma_2.\ \top \vdash \Gamma_1\{c \rightharpoonup c'\}\Gamma_2$, then we have*

1. *$(\forall x \in \mathtt{Asgnd}(c).\ \Gamma_2^\dagger(x) = *) \wedge (\Gamma_1^\dagger(x) = * \Rightarrow \Gamma_2^\dagger(x) = *)$*
2. *$\forall m_1', m_2'.\ [\![c']\!]_{m_1'}(m_2') \neq 0 \implies \Gamma_1^\dagger m_1' = \Gamma_2^\dagger m_2'$*

*Proof.* By structural induction on $c$.

- $c$ is skip: trivial.
- $c$ is $x := e$: When $pc = \top$, $\Gamma_2^\dagger(x) = *$ by the typing rule. $(\Gamma_1^\dagger(x) = * \Rightarrow \Gamma_2^\dagger(x) = *)$ since $\Gamma_2^\dagger(y) = \Gamma_1^\dagger(y)$ for $y \neq x$.
In this case, $c'$ is defined as $(\widehat{x}^\dagger := x + \mathbb{n}^\dagger - e; x := e)$ where $\Gamma_1^\dagger \vdash x : \mathbb{n}^\dagger$. By the semantics, we have $m_2' = m_1'\{[\![x + \mathbb{n}^\dagger - e]\!]_{m_1'}/\widehat{x}^\dagger\}\{[\![e]\!]_{m_1'}/x\}$. Hence, $\Gamma_2 m_2'(x) = m_2'(x) + [\![x + \mathbb{n}^\dagger - e]\!]_{m_1'} = [\![e]\!]_{m_1'} + [\![x + \mathbb{n}^\dagger]\!]_{m_1'} - [\![e]\!]_{m_1'} = \Gamma_1 m_1(x)$.
For any $y \neq x$, both its value and its shadow distance do not change (due to the wellformedness check in rule (T-Asgn)). Hence, $\Gamma_1^\dagger m_1'(y) = \Gamma_2^\dagger m_2'(y)$.
- $c$ is $c_1; c_2$: by induction hypothesis.
- $c$ is if $e$ then $c_1$ else $c_2$: when $pc = \top$, we have $c' = \mathtt{if}\ e\ \mathtt{then}\ (\mathtt{assert}\ (e^\circ); c_1'; c_1'')\ \mathtt{else}\ (\mathtt{assert}\ (\neg e^\circ); c_2'; c_2'')$. By the induction hypothesis, we know that $c_1'$ and $c_2'$ does not modify any shadow variable and their ending typing environments, say $\Gamma_1', \Gamma_2'$, satisfy condition 1. Hence, the ending environment $\Gamma_1' \sqcup \Gamma_2'$ satisfies condition 1 too.
To show (2), we assume $c_1$ is executed without losing generality. Let $m_3'$ be the memory state between $c_1'$ and $c_1''$. By induction hypothesis, $\Gamma_1^\dagger m_1' = (\Gamma_1')^\dagger m_3'$. By the definition of $\Gamma_i, \Gamma_1 \sqcup \Gamma_2, \top \Rrightarrow c_i'', c_1''$ and $c_2''$ only modifies $\widehat{x}^\circ, x \in \mathit{Vars}$. Hence, $\Gamma_2^\dagger m_2' = (\Gamma_1')^\dagger m_3' = \Gamma_1^\dagger m_1'$.
- $c$ is while $e$ do $c'$: by the definition of $\sqcup$ and induction hypothesis, condition 1 holds.
Moreover, by the definition of $\Gamma, \Gamma \sqcup \Gamma_f, \top \Rrightarrow c_s$, $c_s$ only modifies $\widehat{x}^\circ, x \in \mathit{Vars}$. Hence, $\Gamma_2^\dagger m_2' = \Gamma_1^\dagger m_1'$ by induction on the number of iterations.
- $c$ is sampling instruction: does not apply since $pc = \bot$ in the typing rule.

□

**Lemma 11.** *$\forall c, c', c^\dagger, \Gamma_1, \Gamma_2.\ (\forall x \in \mathtt{Asgnd}(c).\ \Gamma_2^\dagger(x) = *) \wedge (\!|c,\Gamma_2|\!)^\dagger = c^\dagger \wedge c'$ is deterministic, and $(\forall m_1', m_2'.\ [\![c']\!]_{m_1'}(m_2') \neq 0 \implies \Gamma^\dagger m_1' = \Gamma^\dagger m_2')$, then we have*

$$\forall m_1', m_2'.\ [\![c'; c^\dagger]\!]_{m_1'}(m_2') = [\![c]\!]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\dagger m_2').$$

*Proof.* $c$ and $c'$ are deterministic. Hence, this lemma can be stated as

$$\forall m_1', m_2'.\ m_2' = [\![c'; c^\dagger]\!]_{m_1'} \implies \Gamma_2^\dagger m_2' = [\![c]\!]_{\Gamma_1^\dagger m_1'}$$

Let $m_3' = [\![c']\!]_{m_1'}$ and $m_2' = [\![c^\dagger]\!]_{m_3'}$. By assumption, we have $\Gamma_2^\dagger m_3' = \Gamma_1^\dagger m_1'$. Moreover, by Lemma 9, we have $\Gamma_2^\dagger m_2' = [\![c]\!]_{\Gamma_2^\dagger m_3'}$. Hence, $\Gamma_2^\dagger m_2' = [\![c]\!]_{\Gamma_2^\dagger m_3'} = [\![c]\!]_{\Gamma_1^\dagger m_1'}$.

□

### D.4 Soundness

Finally, we prove the Pointwise Soundness Lemma.

***Proof of Lemma 3***

*Proof.* By structural induction on $c$. Note that the desired inequalities are trivially true if $[\![c']\!]_{m_1'}(m_2') = 0$. Hence, in the proof we assume that $[\![c']\!]_{m_1'}(m_2') > 0$.

- Case (skip): trivial.
- Case $(x := e)$: by (T-Asgn), we have $\max(c'' \restriction^{m_2'}_{m_1'}) = 0$. This command is deterministic in the sense that when $[\![c']\!]_{m_1'}(m_2') \neq 0$, we have $[\![c']\!]_{m_1'}(m_2') = 1$ and $m_2' = m_1'\{[\![e]\!]_{m_1'}/x\}$. To prove (1) and (2a), it suffices to show that $\Gamma_2^\dagger m_2' = \Gamma_1^\dagger m_1'\{[\![e]\!]_{\Gamma_1^\dagger m_1'}/x\}$ and $\Gamma_2^\circ m_2' = \Gamma_1^\circ m_1'\{[\![e]\!]_{\Gamma_1^\circ m_1'}/x\}$. We prove the latter one as the other (given $pc = \perp$) can be shown by a similar argument.
First, we show $\Gamma_2^\circ m_2'(x) = \Gamma_1^\circ m_1'\{[\![e]\!]_{\Gamma_1^\circ m_1'}/x\}(x)$. Let $\Gamma_1 \vdash e : \langle \mathbb{n}^\circ, \mathbb{n}^\dagger \rangle$. By the typing rule, we have $\Gamma_2^\circ(x) = \mathbb{n}^\circ$, and
$$\begin{aligned}\Gamma_2^\circ m_2'(x) &= m_2'(x) + [\![\mathbb{n}^\circ]\!]_{m_2'} \\ &= [\![e]\!]_{m_1'} + [\![\mathbb{n}^\circ]\!]_{m_2'} \\ &= [\![e]\!]_{m_1'} + [\![\mathbb{n}^\circ]\!]_{m_1'} \\ &= [\![e]\!]_{\Gamma_1^\circ m_1'} \\ &= \Gamma_1^\circ m_1'\{[\![e]\!]_{\Gamma_1^\circ m_1'}/x\}(x).\end{aligned}$$
The third equality is due to well-formedness, and the forth equality is due to Lemma 4.
Second, we show that $\Gamma_2^\circ m_2'(y) = \Gamma_1^\circ m_1'\{[\![e]\!]_{\Gamma_1^\circ m_1'}/x\}(y)$ for $y \neq x$. First, by Rule (T-Asgn), $\Gamma_2^\circ(y) = \Gamma_1^\circ(y)$, $m_1'(y) = m_2'(y)$ and $m_1'(\widehat{y}^\circ) = m_2'(\widehat{y}^\circ)$. If $\Gamma_2^\circ(y) = \Gamma_1^\circ(y) = *$, $\Gamma_2^\circ m_2'(y) = m_2'(y) + m_2'(\widehat{y}^\circ) = m_1'(y) + m_1'(\widehat{y}^\circ) = \Gamma_1^\circ m_1'(y)$. If $\Gamma_2^\circ(y) = \Gamma_1^\circ(y) = \mathbb{n}$, $\Gamma_2^\circ m_2'(y) = m_2'(y) + [\![\mathbb{n}]\!]_{m_2'} = m_1'(y) + [\![\mathbb{n}]\!]_{m_1'} = \Gamma_1^\circ m_1'(y)$, where $[\![\mathbb{n}]\!]_{m_2'} = [\![\mathbb{n}]\!]_{m_1'}$ due to well-formedness.
- Case $(c_1; c_2)$: For any $m_2'$ such that $[\![c_1'; c_2']\!]_{m_1'}(m_2') \neq 0$, there exists some $m'$ such that
$$[\![c_1']\!]_{m_1'}(m') \neq 0 \land [\![c_2']\!]_{m'}(m_2') \neq 0.$$
Let $pc \vdash \Gamma_1\{c_1 \rightharpoonup c_1'\}\Gamma$, $pc \vdash \Gamma\{c_2 \rightharpoonup c_2'\}\Gamma_2$ For (1), we have
$$\begin{aligned}[\![c_1'; c_2']\!]_{m_1'}(m_2') &= \sum_{m'} [\![c_1']\!]_{m_1'}(m') \cdot [\![c_2']\!]_{m'}(m_2') \\ &\leq \sum_{m'} [\![c_1]\!]_{\Gamma_1^\dagger m_1'}(\Gamma^\dagger m') \cdot [\![c_2]\!]_{\Gamma^\dagger m'}(\Gamma_2^\dagger m_2) \\ &\leq \sum_{m'} [\![c_1]\!]_{\Gamma_1^\dagger m_1'}(m') \cdot [\![c_2]\!]_{m'}(\Gamma_2^\dagger m_2') \\ &= [\![c_1; c_2]\!]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\dagger m_2').\end{aligned}$$
Here the second line is by induction hypothesis. The change of variable in the third line is due to Lemma 2.
For (2a) and (2b), let $\epsilon_1 = \max(c_1'' \restriction^{m'}_{m_1'})$, $\epsilon_2 = \max(c_2'' \restriction^{m_2'}_{m'})$ and $\epsilon = \max(c_1''; c_2'' \restriction^{m_2'}_{m_1'})$. Note that $\epsilon_1 + \epsilon_2 \leq \epsilon$ due to the fact that $m_2' \uplus (\epsilon_1 + \epsilon_2) \in [\![c_1''; c_2'']\!]_{m_1' \uplus (0)}$. By induction hypothesis we have one of the following two cases.
  1. $[\![c_2']\!]_{m'}(m_2') \leq \exp(\epsilon_2)[\![c_2]\!]_{\Gamma^\dagger m'}(\Gamma_2^\circ m_2')$. We have
$$\begin{aligned}[\![c_1'; c_2']\!]_{m_1'}(m_2') &= \sum_{m'} [\![c_1']\!]_{m_1'}(m') \cdot [\![c_2']\!]_{m'}(m_2') \\ &\leq \exp(\epsilon_2) \sum_{m'} [\![c_1']\!]_{\Gamma_1^\dagger m_1'}(\Gamma^\dagger m') \cdot [\![c_2']\!]_{\Gamma^\dagger m'}(\Gamma_2^\circ m_2') \\ &\leq \exp(\epsilon_2) \sum_{m'} [\![c_1']\!]_{\Gamma_1^\dagger m_1'}(m') \cdot [\![c_2']\!]_{m'}(\Gamma_2^\circ m_2') \\ &\leq \exp(\epsilon_2)[\![c_1; c_2]\!]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\circ m_2') \\ &\leq \exp(\epsilon)[\![c_1; c_2]\!]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\circ m_2').\end{aligned}$$
  The first inequality is by induction hypothesis. The change of variable in the third line is again due to Lemma 2. The last line is because $\epsilon_1 + \epsilon_2 \leq \epsilon$.
  2. $[\![c_2']\!]_{m'}(m_2') \leq \exp(\epsilon_2)[\![c_2]\!]_{\Gamma^\circ m'}(\Gamma_2^\circ m_2')$. By induction hypothesis on $c_1$, we have two more cases:
     a. $[\![c_1']\!]_{m_1'}(m') \leq \exp(\epsilon_1)[\![c_1]\!]_{m_1^\circ}(m^\circ)$. In this case,
$$\begin{aligned}[\![c_1'; c_2']\!]_{m_1'}(m_2') &= \sum_{m'} [\![c_1']\!]_{m_1'}(m') \cdot [\![c_2']\!]_{m'}(m_2') \\ &\leq \exp(\epsilon) \sum_{m'} [\![c_1]\!]_{\Gamma_1^\circ m_1'}(\Gamma^\circ m') \cdot [\![c_2]\!]_{\Gamma^\circ m'}(\Gamma_2^\circ m_2') \\ &\leq \exp(\epsilon) \sum_{m'} [\![c_1]\!]_{\Gamma_1^\circ m_1'}(m') \cdot [\![c_2]\!]_{m'}(\Gamma_2^\circ m_2') \\ &\leq \exp(\epsilon)[\![c_1; c_2]\!]_{\Gamma_1^\circ m_1'}(\Gamma_2^\circ m_2').\end{aligned}$$
     The second line is by induction hypothesis and the fact that $\epsilon_1 + \epsilon_2 \leq \epsilon$. The change of variable in the third line is due to Lemma 2.
     b. $[\![c_1']\!]_{m_1'}(m') \leq \exp(\epsilon_1)[\![c_1]\!]_{\Gamma_1^\dagger m_1'}(\Gamma^\circ m')$. We have
$$[\![c_1'; c_2']\!]_{m_1'}(m_2') \leq \exp(\epsilon)[\![c_1; c_2]\!]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\circ m_2')$$
     by a similar argument as above.
- Case (if $e$ then $c_1$ else $c_2$): If $\Gamma_1 \vdash e$ : bool, then by Lemma 5 we have $[\![e]\!]_{m_1'} = [\![e]\!]_{\Gamma_1^\circ m_1'} = [\![e]\!]_{\Gamma_1^\dagger m_1'}$. Hence, the same branch is taken in all related executions. By rule (T-If), the transformed program is

$$\texttt{if}\ e\ \texttt{then}\ (\texttt{assert}\ ((\!|e|\!)^\circ); c_1'; c_1'')\ \texttt{else}\ (\texttt{assert}\ (\neg(\!|e|\!)^\circ); c_2'; c_2'')$$

  and $\Gamma_1, \Gamma_1 \sqcup \Gamma_2, \perp \Rrightarrow c_1''$, $\quad \Gamma_2, \Gamma_1 \sqcup \Gamma_2, \perp \Rrightarrow c_2''$. Without loss of generality, suppose that $c_1$ is executed in all related executions. By Lemma 7, there is a unique $m'$ such that $[\![c_1']\!]_{m_1'}(m') \neq 0$. By induction hypothesis, we have
$$[\![c_1']\!]_{m_1'}(m') \leq [\![c_1]\!]_{\Gamma^\dagger m_1'}(\Gamma_1^\dagger m')$$
  Moreover, by Lemma 7, $\Gamma_1^\dagger m' = \Gamma_1^\dagger m_2'$ and
$$[\![c_1'; c_1'']\!]_{m_1'}(m_2') = [\![c_1']\!]_{m_1'}(m').$$
  Hence, we have
$$[\![c_1'; c_1'']\!]_{m_1'} m_2' \leq [\![c_1]\!]_{\Gamma^\dagger m_1'}(\Gamma_1^\dagger m') = [\![c_1]\!]_{\Gamma^\dagger m_1'}(\Gamma_1^\dagger m_2')$$
  (2a) or (2b) can be proved in a similar way.

When $\Gamma_1 \nvdash e : \texttt{bool}$, the aligned execution will still take the same branch due to the inserted assertions. Hence, the argument above still holds for (2a) or (2b).

For the shadow execution, we need to prove (1). By rule (T-If), $pc' = \top$, and the transformed program is

$$\texttt{if } e \texttt{ then } (\texttt{assert } ((\!|e|\!)^\circ); c_1'; c_1'') \texttt{ else } (\texttt{assert } (\neg (\!|e|\!)^\circ); c_2'; c_2'');$$
$$(\!|\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \Gamma_1 \sqcup \Gamma_2|\!)^\dagger$$

By Lemma 10 and the definition of $\Rrightarrow$ under high pc, we have that $\forall m_1', m_2'.\ [\![c']\!]_{m_1'}(m_2') \neq 0 \Rightarrow \Gamma_2^\dagger m_1' = \Gamma_1^\dagger m_1'$, and $\Gamma_1 \sqcup \Gamma_2(x) = * \ \forall x \in \texttt{Asgnd}(c_1; c_2)$. Furthermore, the program is deterministic since it type-checks under $\top$. Therefore, $\max(W'' \restriction_{m_1}^{m_2}) = 0$ and (1) holds by Lemma 11.

- Case ($\texttt{while } e \texttt{ do } c$): Let $W = \texttt{while } e \texttt{ do } c$. If $\Gamma_1 \vdash e : \texttt{bool}$, then $[\![e]\!]_{m_1'} = [\![e]\!]_{\Gamma_2^\circ m_1'} = [\![e]\!]_{\Gamma_2^\dagger m_1'}$. By rule (T-While) we have $pc' = \bot$, $\Gamma_2 = \Gamma \sqcup \Gamma_f$ and the transformed program is
$$W' = c_s;\ \texttt{while } e \texttt{ do } (\texttt{assert } ((\!|e|\!)^\circ); c'; c'')$$
where $\bot \vdash \Gamma_1 \sqcup \Gamma_f \{c \rightharpoonup c'\}\ \Gamma_f$, $(\Gamma_1, \Gamma_1 \sqcup \Gamma_f, \bot) \Rrightarrow c_s$ and $(\Gamma_f, \Gamma_1 \sqcup \Gamma_f, \bot) \Rrightarrow c''$. We proceed by natural induction on the number of loop iterations (denoted by $i$).
When $i = 0$, we have $[\![e]\!]_{m_1'} = \texttt{false}$. By the semantics, we have $[\![W']\!]_{m_1'} = \texttt{unit}\,([\![c_s]\!]_{m_1'})$, $[\![W]\!]_{\Gamma_1^\circ m_1'} = \texttt{unit}\,(\Gamma_1^\circ m_1')$, and $[\![W]\!]_{\Gamma_1^\dagger m_1'} = \texttt{unit}\,(\Gamma_1^\dagger m_1')$. Furthermore, $\max(W'' \restriction_{m_1}^{m_2}) = 0$. By Lemma 7, $(\Gamma_1^\circ m_1') = (\Gamma_2^\circ([\![c_s]\!]_{m_1'}))$ and $(\Gamma_1^\dagger m_1') = (\Gamma_2^\dagger([\![c_s]\!]_{m_1'}))$. So desired result holds.
When $i = j + 1 > 0$, we have $[\![e]\!]_{m_1'} = \texttt{true}$. By the semantics, we have $[\![W']\!]_{m_1'} = [\![c_s; \underbrace{c'; c''}_{j}; c'; c'']\!]_{m_1'}$ and $[\![W]\!]_{\Gamma_1^\star m_1'} = [\![c_s; \underbrace{c'; c''}_{j}; c'; c'']\!]_{\Gamma_1^\star m_1'}, \star \in \{\circ, \dagger\}$. For any $m'$ such that $[\![c_s; \underbrace{c'; c''}_{j};]\!]_{m_1'}(m') = k_1 \neq 0$ and $[\![c'; c'']\!]_{m'}(m_2') = k_2 \neq 0$, we know that $[\![\underbrace{\ c\ }_{j}]\!]_{\Gamma_1^\star m_1'}(\Gamma_2^\star m') = k_1$. Moreover, there is a unique $m_3'$ such that $[\![c']\!]_{m'}(m_3') = k_2 \wedge [\![c'']\!]_{m_3'}(m_2') = 1$ by Lemma 7. By structural induction on $c$, we have $k_1 \leq [\![c]\!]_{\Gamma_2^\dagger m'}(\Gamma_f^\dagger m_3')$ and either $k_1 \leq \exp(\max(c' \restriction_{m_1'}^{m_2'}))[\![c]\!]_{\Gamma_2^\circ m'}(\Gamma_f^\circ m_3')$ or $k_1 \leq \exp(\max(c'' \restriction_{m_1'}^{m_2'}))[\![c]\!]_{\Gamma_2^\dagger m'}(\Gamma_f^\circ m_3')$. By Lemma 7, $\Gamma_f^\star m_3' = \Gamma_2^\star m_2'$. Hence, we can replace $\Gamma_f$ with $\Gamma_2$ in the inequalities. Finally, the desired result holds following the same argument as in the composition case.
Otherwise, by rule (T-While) we have $pc' = \top$, $\Gamma_2 = \Gamma \sqcup \Gamma_f$ and the transformed program is $W' = c_s; W''$ where
$$W'' = \texttt{while } e \texttt{ do } (\texttt{assert } ((\!|e|\!)^\circ); c'; c'');\ (\!|\texttt{while } e \texttt{ do } c, \Gamma_2|\!)^\dagger$$
and $\top \vdash \Gamma_1 \sqcup \Gamma_f \{c \rightharpoonup c'\}\ \Gamma_f$, $(\Gamma_1, \Gamma_1 \sqcup \Gamma_f, \top) \Rrightarrow c_s$ and $(\Gamma_f, \Gamma \sqcup \Gamma_f, \top) \Rrightarrow c''$. Since we do not have sampling instructions in this case, the program is deterministic, thus $\max(W'' \restriction_{m_1}^{m_2}) = 0$. Note that by rule (T-Asgn), $\forall x \in \texttt{Asgnd}(c)$, $\Gamma_f^\dagger(x) = *$. Hence, $\Gamma_2^\dagger(x) = *$ for $x \in \texttt{Asgnd}(c)$. Morevoer, let $m'$ be the unique memory such that $[\![c_s]\!]_{m_1'}(m') \neq 0$, we have $\Gamma_1^\dagger m_1' = \Gamma_2^\dagger m'$ by Lemma 7. Therefore, by Lemmas 10 and 11, we have
$$[\![W']\!]_{m_1'}(m_2') = [\![W'']\!]_{m'}(m_2') = [\![W]\!]_{\Gamma_2^\dagger m'}(\Gamma_2^\dagger m_2') = [\![W]\!]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\dagger m_2')$$

- Case ($\eta := \texttt{Lap } r; \mathcal{S}, \mathbb{n}_\eta$): let $\mu_r$ be the probability density (resp. mass) function of the Laplace (resp. discrete Laplace) distribution with zero mean and scale $r$. Since $\mu_r(v) \propto \exp(-|v|/r)$ we have
$$\forall v, d \in \mathbb{R},\ \mu_r(v) \leq \exp(|d|/r)\mu_r(v + d). \qquad (4)$$
When $[\![c']\!]_{m_1'}(m_2') \neq 0$, we have $m_2' = m_1'\{\mathsf{v}/\eta\}$ for some constant $\mathsf{v}$, and $[\![c']\!]_{m_1'}(m_2') = \mu_r(\mathsf{v})$.
We first consider the case $\mathcal{S} = \circ$. By (T-Laplace), we have $\Gamma_2(x) = \Gamma_1(x)$ for $x \neq \eta$, and $\Gamma_2(\eta) = \langle \mathbb{n}_\eta, 0\rangle$. Therefore, for $x \neq \eta$, we have
$$\begin{aligned}\Gamma_2^\dagger m_2'(x) &= m_2'(x) + [\![\Gamma_2^\dagger(x)]\!]_{m_2'}\\ &= m_1'(x) + [\![\Gamma_1^\dagger(x)]\!]_{m_2'}\\ &= m_1'(x) + [\![\Gamma_1^\dagger(x)]\!]_{m_1'}\\ &= \Gamma_1^\dagger m_1'(x).\end{aligned}$$
In the second equation, we have $m_2'(x) = m_1'(x)$ because $m_2 = m_1\{\mathsf{v}/\eta\}$ and $x \neq \eta$. The third equation is due to the well-formedness assumption. Also, $\Gamma_2^\dagger m_2'(\eta) = m_2'(\eta) = \mathsf{v}$ since $\Gamma_2^\dagger(\eta) = 0$. Therefore, we have $\Gamma_2^\dagger m_2' = \Gamma_1^\dagger m_1'\{\mathsf{v}/\eta\}$ and thus $[\![c]\!]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\dagger m_2') = \mu_r(\mathsf{v}) = [\![c']\!]_{m_1'}(m_2')$.
Similarly, we can show that $\Gamma_2^\circ m_2' = \Gamma_1^\circ m_1'\{\mathsf{v} + \mathsf{d}/\eta\}$ and therefore $[\![c]\!]_{\Gamma_1^\circ m_1'}(\Gamma_2^\circ m_2') = \mu_r(\mathsf{v} + \mathsf{d})$. Furthermore, since $\mathbf{v}_\epsilon := \mathbf{v}_\epsilon + |\mathbb{n}_\eta|/r$, the set $(c'' \restriction_{m_1}^{m_2})$ contains a single element $[\![|\mathbb{n}_\eta|/r]\!]_{m_2} = |\mathsf{d}|/r$, and thus $\max(c'' \restriction_{m_1}^{m_2}) = |\mathsf{d}|/r$. Therefore, we have
$$\begin{aligned}[\![c']\!]_{m_1'}(m_2') &= \mu_r(\mathsf{v})\\ &\leq \exp(|\mathsf{d}|/r)\mu_r(\mathsf{v} + \mathsf{d})\\ &= \exp(\max(c'' \restriction_{m_1}^{m_2}))[\![c]\!]_{\Gamma_1^\circ m_1}(\Gamma_2^\circ m_2).\end{aligned}$$
The second inequality is due to (4). Thus (1) and (2a) hold.
Next consider the case $\mathcal{S} = \dagger$. By the typing rule, we have $\Gamma_2^\circ(x) = \Gamma_2^\dagger(x) = \Gamma_1^\dagger(x)$ for $x \neq \eta$, and $\Gamma_2(\eta) = \langle \mathbb{n}_\eta, 0\rangle$. By a similar argument we have $\Gamma_2^\dagger m_2' = \Gamma_1^\dagger m_1'\{\mathsf{v}/\eta\}$ and $\Gamma_2^\circ m_2' = \Gamma_1^\dagger m_1'\{\mathsf{v} + \mathsf{d}/\eta\}$. Furthermore, since $\mathbf{v}_\epsilon := 0 + |\mathbb{n}_\eta|/r$, we have $\max(c'' \restriction_{m_1'}^{m_2'}) = |\mathsf{d}|/r$. Therefore, we have
$$\begin{aligned}[\![c']\!]_{m_1'}(m_2') &= \mu_r(\mathsf{v}) = [\![c]\!]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\dagger m_2')\\ &\leq \exp(|\mathsf{d}|/r)\mu_r(\mathsf{v} + \mathsf{d})\\ &= \exp(\max(c'' \restriction_{m_1'}^{m_2'}))[\![c]\!]_{\Gamma_1^\dagger m_1'}(\Gamma_2^\circ m_2').\end{aligned}$$
Thus (1) and (2b) hold.
Finally, consider $\mathcal{S} = e\ ?\ \mathcal{S}_1 : \mathcal{S}_2$. For any $m_1'$, we have $\mathcal{S} = \mathcal{S}_1$ if $[\![e]\!]_{m_1'} = \texttt{true}$ and $\mathcal{S} = \mathcal{S}_2$ if $[\![e]\!]_{m_1'} = \texttt{false}$. By

induction, $\mathcal{S}$ evaluates to either $\circ$ or $\dagger$ under $m_1'$. Thus the proof follows from the the two base cases above.

□

## E Formal Semantics for the Target Language

The denotational semantics interprets a command $c$ in the target language as a function $[\![c]\!] : \mathcal{M} \to \mathcal{P}(\mathcal{M})$. The semantics of commands are formalized as follows.

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