Abstract
Recent work on formal verification of differential privacy shows a trend toward usability and expressiveness – generating a correctness proof of sophisticated algorithms 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 \( \text{Dist}(A) \), as the set of functions \( \mu : A \rightarrow [0, 1] \), such that \( \sum_{a \in A} \mu(a) \leq 1 \). When applied to an event \( E \subseteq A \), we define \( \mu(E) \triangleq \sum_{e \in E} \mu(e). \)

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

---

**Definition 1** (Pure Differential privacy). Let \( \epsilon \geq 0 \). A probabilistic computation \( M : A \rightarrow B \) 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 \( f : \mathbb{R} \rightarrow \mathbb{R} \) 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:

\[
P(M(a_1) \in E) = \sum_{\eta: a_1 + \eta \in E} p_{1/\epsilon}(\eta) \leq \sum_{\eta: a_2 + f(\eta) \in E} p_{1/\epsilon}(\eta) \leq e^{\epsilon} \sum_{\eta: a_2 + \eta \in E} p_{1/\epsilon}(\eta) = e^{\epsilon} P(M(a_2) \in E).
\]

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 \( p_{1/\epsilon}(\eta + c) \leq e^{\epsilon} \), when the Laplace distribution has scale \( 1/\epsilon \).

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 \( f : \mathbb{R}^n \rightarrow \mathbb{R}^n \) 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
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 $\alpha_0^{(i)} = 1, \alpha_1^{(i)} = 2, \alpha_2^{(i)} = 1$, respectively. In this case, the noisy query answers are $q[0] + \alpha_0^{(i)} = 2, q[1] + \alpha_1^{(i)} = 4, q[2] + \alpha_2^{(i)} = 3$ 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 a injective function of random bits in $D_1$ to random bits in $D_2$ to make the output the same. Recall that $\alpha_0^{(i)}, \alpha_1^{(i)}, \alpha_2^{(i)}$ 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 $q[i] + \alpha_1^{(i)} + 1$ on $D_1$ for any $i$. By the adjacency assumption, that is the same as $q[1] + \alpha_1^{(2)}$ is greater than any $q[i] + \alpha_1^{(i)}$ 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.
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 \(\alpha_0^{(1)}, \alpha_1^{(1)}\) and so on. Note that the shadow execution uses \(\alpha_i^{(1)} = \alpha_i^{(1)}\) 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., \(\text{max}\) 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., \(\text{max}\) 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 \(\alpha_i^{(1)} + 2 = 4\) 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 \(\text{num}(\eta, \rho)\), where \(\eta^0\) and \(\rho^\dagger\) represent the “difference” of its value in the aligned and shadow execution respectively. In Figure 1, non-private variables, such as \(\epsilon, \text{size}\), 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 \(\ast\), which is desugared to a special variable \(\widetilde{q}^\dagger[i]\) (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 \(v_r\). The soundness of the type system ensures the following property: if \(v_r\) 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.
we only consider the most interesting random expression.

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 $\eta \leftarrow g$.

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, Lap $r$. Semantically, $\eta \leftarrow r$ 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 $\mathcal{B}_{(\mathcal{d}_r, \mathcal{d}_s)}$, where $\mathcal{B}$ is the base type, and $\mathcal{d}_r, \mathcal{d}_s$ represent the alignments for the execution on adjacent database and shadow execution respectively. Base type is standard: it includes $\text{num}$ (numeric type), $\text{bool}$ (Boolean), or a list of elements with type $\tau$ ($\mathcal{d}_s$).

Distance $d$ 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 $\text{bool}$, list $\tau$ and $r_1 \rightarrow r_2$) are always associated with $(0, 0)$, hence omitted in the syntax. Note that this does not rule out numeric distances in nested types. For example, $(\text{list} \text{num}_{(1, 1)})$ stores numbers that differ by exactly one in both aligned and shadow executions.

Distance $d$ can either be a numeric expression ($n$) in the language or $\star$. A variable $x$ with type num($\langle \tau, e \rangle$) is desugared as $x : \Sigma(\langle \tilde{x}, \tilde{\tau} \rangle : \text{num}_{\langle \langle \Omega \rangle, 1 \rangle}) : \text{num}_{\langle \langle \Omega \rangle, 1 \rangle}$, where $\tilde{x}$, $\tilde{\tau}$ 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 list $\text{num}_{\langle \tau, e \rangle}$, along with the precondition $\forall i \geq 0. -1 \leq q[i] \leq 1$. 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 $\star$; the type system instruments the source code to maintain the correct values of $\tilde{x}$, $\tilde{\tau}$ (Section 4).

### Sampling with selectors

Each sampling instruction is attached with a few annotations for proving differential privacy, in the form of $(\eta := \text{Lap } r, S, \nu\eta)$. Note that just like types, the annotations $S, \nu\eta$ have no effects on the program semantics; they only show up in verification. Intuitively, a selector $S$ picks a version ($k \in \left\{\circ, \dag\right\}$) for all program variables (including the previously sampled variables) at the sampling instruction, as well as constructs randomness alignment for $\eta$ specified by $\nu\eta$ (note that the distance cannot be $\star$ by syntactical restriction here). By definition, both $S$ and $\nu\eta$ 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 $(\eta := \text{Lap } (2/e), \Omega ? \dag : \circ, \Omega ? 2 : 0)$, where $\Omega$ is $q[i] + \eta > bq \lor i = 0$, the condition when a new max is found.

### 3.2 Semantics

As standard, the denotational semantics of the probabilistic language is defined as a mapping from initial memory to a distribution on (possible) final outputs. Formally, let $M$ be a set of memory states where each $m \in M$ maps all (normal and random) variables ($NVars \cup RVars$) to their values.

The semantics of an expression $e$ of base type $B$ is interpreted as a function $\llbracket e \rrbracket : M \rightarrow [\mathcal{B}]$, where $[\mathcal{B}]$ represents the set of values belonging to the base type $B$. We omit
expression semantics since it is standard. A random expression \( q \) is interpreted as a distribution on real values. Hence, \([q] : \text{Dist}([\text{num}]) \). Moreover, a command \( c \) is interpreted as a function \([c] : M \rightarrow \text{Dist}(M)\). For brevity, we write \([e]_\text{m}\) and \([c]_\text{m}\) instead of \([e](m)\) and \([c](m)\) hereafter. Finally, all programs have the form \((c; \text{return } e)\) where \( c \) contains no return statement. A program is interpreted as a function \( m \rightarrow \text{Dist}([M]) \) where \( B \) 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 \( \Gamma(x) = (d_\text{v}, d_\text{f}) \) for \( \exists B. \Gamma(x) = B(c_\text{v}, c_\text{f}) \) when the base type \( B \) 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:

\[
\Psi \Rightarrow (\lambda_1 \circ \lambda_2 \iff (\lambda_1 + m_1) \circ (\lambda_2 + m_3) \wedge (\lambda_1 + m_2) \circ (\lambda_2 + m_4))
\]

This constraint states that the boolean value of \( \lambda_1 \circ \lambda_2 \) 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 \( \lambda_1 \circ \lambda_2 \) has distances \( (0, 0) \).

4.3 Commands

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

\[
\Gamma \vdash \Gamma(c \rightarrow c') \Gamma'
\]

meaning that starting from the previous typing environment \( \Gamma' \), the new typing environment is \( \Gamma' \) after \( c \). We will discuss the other components \( \Gamma c \) 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 \( \Gamma \) for now since only aligned variables are discussed.

Flow-Sensitivity In each typing rule \( \Gamma \vdash \Gamma(c \rightarrow c') \Gamma' \), 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, \( v'^e \), and updates the distance of \( x \)'s value after assignment to \( v' \).)

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.

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

Rule (T-WHILE) is similar, except that it requires a fixed point \( \Gamma_f \) such that \( \Gamma \vdash \Gamma \sqcup \Gamma_f(c) \Gamma_f \). In fact, this rule is determined since we can construct the fixed point as follows (the construction is similar to the one in [27]):

\[
\Gamma \vdash \Gamma_f(c \rightarrow c')^{i} \Gamma_f^{i''} \text{ for all } 0 \leq i \leq n
\]

where \( \Gamma_f = \Gamma_0, \Gamma_f^{i} = \Gamma_f^{i+1} \sqcup \Gamma_f, \Gamma_f^{i+1} = \nu \Gamma_f \) and \( \nu \Gamma_f \sqcup \Gamma_f^{i} \). It is easy to check that \( \Gamma_f = \Gamma_f^{n+1} = \nu \Gamma_f \sqcup \Gamma_f^{i} \). Hence, \( \Gamma_f^{n} \) is a fixed point: \( \nu \Gamma_f \sqcup \Gamma_f^{i} \). Moreover, the computation above always terminates since all typing rules are monotonic on typing environments and the lattice has a height of 2.

Maintaining dynamically tracked distances Each typing rule \( \Gamma \vdash \Gamma(c \rightarrow c') \Gamma' \) also sets the value of \( \hat{\Omega} \) to maintain distance dynamically whenever \( \Gamma(x) = \ast \). 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-IR) generates \& type when \( \Gamma_1(x) \neq \Gamma_2(x) \).

In this case, we use the auxiliary instrumentation rule in the
Typing rules for expressions

\[\Gamma \vdash r : \text{num}(0, 0) \quad \text{(T-Num)}\]
\[\Gamma \vdash b : \text{bool} \quad \text{(T-Boolean)}\]
\[\Gamma \vdash e_1 : \text{num}(r_1, r_2) \quad \Gamma \vdash e_2 : \text{num}(r_3, r_4) \quad \text{(T-PLUS)}\]
\[\Gamma \vdash e_1 \odot e_2 : \text{num}(r_5, r_6, r_7) \quad \text{(T-OPLUS)}\]
\[\Gamma \vdash e : \text{bool} \quad \text{(T-BOOLO)}\]
\[\Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e : \text{list} \tau \quad \text{(T-TUPLE)}\]
\[\Gamma \vdash e : \text{list} \tau \quad \Gamma \vdash e_1 :: e_2 : \tau \quad \text{(T-CONS)}\]
\[\Gamma \vdash e_1 :: e_2 : \tau \quad \text{(T-CONS)}\]

Typing rules for commands

\[\Gamma \vdash e : \mathcal{B}(v_1, v_2) \quad \langle \Gamma', c \rangle = \begin{cases} 
(\langle \Gamma \mapsto \mathcal{B}(v_1, v_2) \rangle, \text{skip}), & \text{if } pc = \bot \\
(\langle \Gamma \mapsto \mathcal{B}(v_1, v_2) \rangle, \overline{w} = x \mapsto v' - e), & \text{else}
\end{cases} \quad \text{(T-ASSIGN)}\]
\[pc + \Gamma [\text{skip} \mapsto \text{skip}] \Gamma \quad \text{(T-Skip)}\]
\[pc + \Gamma [c_1 \mapsto c_2]' \Gamma_1 \quad pc + \Gamma [c_2 \mapsto c_2]' \Gamma_2 \quad pc + \Gamma [c_1 \mapsto c_2]' \Gamma_1 \Gamma_2 \quad \text{(T-SEQ)}\]
\[pc + \Gamma \{c_1 \mapsto c_2\}' \Gamma_1 \Gamma_2 \quad pc + \Gamma [\text{if } e \text{ then } c_1 \text{ else } c_2 \mapsto c_2]' \Gamma_1 \Gamma_2 \quad \text{(T-IF)}\]
\[pc + \Gamma \{\text{while } e \text{ do } c \mapsto c\}' \Gamma_1 \Gamma_2 \quad \text{(T-WHILE)}\]

Typing rules for random assignments

\[pc = \bot \quad \Gamma' = \lambda x. \langle S(v_1, v_2) \rangle \quad \text{where } \Gamma + x : \mathcal{B}(v_1, v_2) \quad \Psi \Rightarrow (\eta + \eta) / \eta = (\eta + \eta) / \eta \Rightarrow \eta_1 = \eta_2 \quad \text{(T-LAPLACE)}\]

Instrumentation rule

\[c^o = \{x^o := v | \Gamma_1(x) = \text{num}(v, d_0) \land \Gamma_2(x) = \text{num}(v, d_0)\} \quad c^+ = \{x^+ := v | \Gamma_1(x) = \text{num}(v, e_2) \land \Gamma_2(x) = \text{num}(v, e_2)\} \quad \text{(T-EXEC)}\]

Select function

\[o((e_1, e_2)) = e_1 \quad \text{and } ((e_1, e_2)) = e_2 \quad \text{if } e \in S_1 : S_2 \quad \text{then } s_1(e_1, e_2) = e \quad \text{if } e \text{ otherwise } \quad \text{(T-EXEC)}\]

PC update function

\[\text{updPC}(pc, \Gamma, e) = \begin{cases} \bot, & \text{if } pc = \bot \land \Psi \Rightarrow (e \mapsto (e, \Gamma)') \\
\top, & \text{else}
\end{cases} \quad \text{(T-EXEC)}\]

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 $\Gamma_1, \Gamma_2, pc \Rightarrow c'$, assuming $\Gamma_1 \subseteq \Gamma_2$. In particular, for each variable $x$ whose distance is "upgraded" to $\ast$, the rule sets $\tilde{x}$ to the distance previously tracked by the type system ($\Gamma(x)$). 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 $\Gamma_1 : \{x : 1, y : 0\}$. 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 $\Gamma_1 \cup \Gamma_2 = \{x : \ast, y : 0\}$. Since $\Gamma_1(x) \neq \Gamma_2(x)$, instrumentation rule is also applied, which instruments $\tilde{x} := 0$ after $x := y$ and $\tilde{x} := 1$ after $y := 1$.

Rule (T-While) may also generate $\ast$ 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 $c_1$ is inserted before loop and $c_2$ 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 $\Gamma^o(y) = x$ and $m(x) = 1$. Since this assignment does not modify the value of $y$, the aligned value of $y$ (i.e., $y + \Gamma^o(y)$) should not change. However, $\Gamma^o(y)$ changes from 1 to 2 after the assignment.

To avoid this issue, we check the following condition for each assignment $x := e$: $\forall y \in \text{Vars}. x \notin \text{Vars}(\Gamma(y))$. In case that the check fails for some $y$, we promote its distance to $\ast$, and use the auxiliary instrumentation $\Rightarrow$ to set $\tilde{y}^o$ 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, $(\cdot, \Gamma)^o$ 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 $\bot$ and $\top$ (introducing program counters in a type system is common in information flow control to track implicit flows [37]). Here, $\top$ (resp. $\bot$) means that the shadow execution might take a different branch (resp. must take the same branch) as the original execution.

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

Take a branch $(\text{if } e \text{ then } c_1 \text{ else } c_2)$ for example. The transition happens when $pc = \bot \land pc' = \top$. 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., $x + \tilde{x}$), 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 $\eta := \text{Lap} \ r, \ S$ in ShadowDP. Here, the selector $S$ and numeric distance $\nu_\eta$ are annotations provided by a programmer to aid type checking. For the sample $\eta$, the aligned distance is specified by $\nu_\eta$ 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 $\text{num}(\nu_\eta, 0)$.

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 $S((e_1, e_2))$ 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):

$$
\{\text{bq} : \{\ast, \circ, \ast\}, \cdots\}
\eta := \text{Lap} \ (2/e), \Omega \ ? \ : \circ, \Omega \ ? \ : 0;
\{\text{bq} : \{\tilde{\eta}, \cdots\}, \nu_\eta : (\Omega \ ? \ : 0, 0), \cdots\}
$$
\[ \eta := \text{Lap } r, S, r_\eta \] \[ \text{havoc } \eta; v_\epsilon := S((v_\epsilon, 0)) + |r_\eta|/r; \]
\[ c \Rightarrow c, \text{ if } c \text{ is not a sampling command} \]

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

In this example, \( S \) is \( \Omega \)?^†\( \circ \). So the aligned distance of variable \( bq \) will be \( \Omega\hat{b}q^\dagger : \hat{b}q^\circ \), the shadow distance of variable \( bq \) is still \( \hat{b}q^\dagger \). The aligned distance of \( \eta \) is \( \Omega \)? : 0, 0), 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 \texttt{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 which sets the variable \( \eta \) to an arbitrary value upon execution, as shown in Figure 5.

Note that a distinguished variable \( v_\epsilon \) 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 \( r_\eta \) is associated with a privacy cost of \( |r_\eta|/r \). The reason is that the ratio of any two points that are \( |r_\eta| \) apart in the Laplace distribution with scaling factor \( r \) is bounded by \( \exp(|r_\eta|/r) \). 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., \( S \) selects \( \dagger \)): the rule sets privacy cost to \( 0 + |r_\eta|/r \) in this case.

In Figure 1, \( v_\epsilon \) is set to \( \Omega \)? : \( 0 : v_\epsilon + \Omega \)? : 0 which is the same as \( \Omega \)? : \( v_\epsilon \). 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 \( v_\epsilon \).

The type system guarantees the following important property: if the original program type checks and the privacy cost \( v_\epsilon \) 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 \( v_\epsilon \leq \epsilon \) 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:

\[
\text{pc} \vdash \Gamma_1 \{ c \rightarrow c' \} \Gamma_2 \quad \text{and} \quad c' \Rightarrow c''
\]

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 \( v_\epsilon \) 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 \( v_\epsilon \) 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 \( \hat{\text{Vars}} = \bigcup_{x \in \text{Vars}} (\hat{x}, \hat{\eta}) \) and introduce a distance environment \( \gamma : \hat{\text{Vars}} \rightarrow \hat{\mathbb{R}} \).

**Definition 2.** Let \( \gamma : \hat{\text{Vars}} \rightarrow \mathbb{R} \). For any \( m \in \mathcal{M} \), there is an extension of \( m \), written \( m \triangledown (\gamma) \), such that

\[
m \triangledown (\gamma)(x) = \begin{cases} m(x), & x \in \text{Vars} \\ \gamma(x), & x \in \hat{\text{Vars}} \end{cases}
\]

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 \( \text{pc} \vdash \Gamma_1 \{ c \rightarrow c' \} \Gamma_2 \). 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. \( \square \)

From now on, we will use \( m'_2 \) to denote the unique extension of \( m_2 \) satisfying the property above.

#### Γ-Relation

To formalize and prove the soundness property, we notice that a typing environment \( \Gamma \) along with distance environment \( \gamma \) induces two binary relations on memories. We write \( m_1 \triangledown (\gamma) \Gamma^0 m_2 \) (resp. \( m_1 \triangledown (\gamma) \Gamma^1 m_2 \)) when \( m_1, m_2 \) are related by \( \Gamma^0 \) (resp. \( \Gamma^1 \)) 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 \( \epsilon, \text{size} \) 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 \( m_1 \otimes (\gamma) \).

**Definition 3** (\( \Gamma \)-Relations). Two memories \( m_1' \) (in the form of \( m_1 \otimes (\gamma) \)) and \( m_2 \) are related by \( \Gamma_\to \), written \( m_1' \Gamma \to m_2 \), if \( \forall x \in \text{Vars} \cup \Gamma \text{Vars} \), we have

\[
m_2(x) = m_1'(x) + m_1'(d^x) \text{ if } \Gamma \vdash x : \text{num}(c', d')
\]

We define the relation on non-numerical types and the \( \Gamma \) relation in a similar way.

By the definition above, \( \Gamma \to \) introduces a function from \( M' \to M \). Hence, we use \( \Gamma_\to m_1' \) as the unique \( m_1' \) such that \( m_1' \Gamma \to m_2 \). The \( \Gamma \) counterparts are defined similarly.

**Injectivity** For alignment-based proofs, given any \( \gamma \), both \( \Gamma_\to \) and \( \Gamma_\to \) must be injective functions [42]. The injectivity of \( \Gamma \) over the entire memory follows from the injectivity of \( \Gamma \) over the random noises \( \eta \in \Gamma \text{Vars} \), which is checked as the following requirement in Rule (T-LAPLACE):

\[
\Psi (\equiv (\eta + v_\eta)|\eta_1/\eta = (\eta + v_\eta)|\eta_2/\eta) \Rightarrow \eta_1 = \eta_2)
\]

where all variables are universally quantified. Intuitively, this is true since the non-determinism of the program is purely from that of \( \eta \in \Gamma \text{Vars} \).

**Lemma 2** (Injectivity). Given \( c, c', p, m', m_1', m_2', \Gamma_1, \Gamma_2 \) such that \( p \Gamma \Gamma_1[c \rightarrow c'] \Gamma_2 \), \( \|c\|_{m_1'} \neq 0 \land \|c\|_{m_2'} \neq 0 \), \( v \in \{\circ, \hat{\circ} \} \), then we have

\[
\Gamma_2^* m_1 = \Gamma_1^* m_2 \implies m_1' = m_2'
\]

**Soundness** The soundness theorem connects the “privacy cost” of the probabilistic program to the distinguished variable \( v \) in the target program \( c'' \). To formalize the connection, we first extend memory one more time to include \( v \):}

**Definition 4.** For any extended memory \( m' \) and constant \( \epsilon \), there is an extension of \( m' \), written \( m' \otimes (\epsilon) \), so that \( m' \otimes (\epsilon)v = \epsilon \), and \( m' \otimes (\epsilon)x = m(x) \), \( \forall x \in \text{dom}(m') \).

For a transformed program and a pair of initial and final memories \( m_1' \) and \( m_2' \), we identify a set of possible \( v_e \) 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 \( c' \Rightarrow c'' \), \( m_1' \) and \( m_2' \), the consistent costs of executing \( c'' \) wrt. \( m_1' \) and \( m_2' \), written \( \text{cost}_{m_1'}(c'') \), is defined as

\[
\text{cost}_{m_1'}(c'') \equiv \{ \epsilon \mid m_2' \otimes (\epsilon) \in \text{cost}(c'')_{m_1'(0)} \}
\]

Since \( \text{cost}_{m_1'}(c'') \) by definition is a set of values of \( v_e \), we write \( \max(\text{cost}_{m_1'}(c'')) \) 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 \( p \Gamma \Gamma_1[c \rightarrow c'] \Gamma_2 \land c' \Rightarrow c'' \), then \( \forall m_1', m_2' : 

(i) the following holds:

\[
\|c''\|_{m_1'}(m_2') \leq \|c\|_{m_1'}(\Gamma_2^* m_2') \quad \text{when } pc = \perp
\]

(ii) one of the following holds:

\[
\begin{align*}
\|c''\|_{m_1'}(m_2') &\leq \exp(\text{max}(c'' \mid m_2')) \|c\|_{m_1'}(\Gamma_2^* m_2') \quad \text{(2a)} \\
\|c''\|_{m_1'}(m_2') &\leq \exp(\text{max}(c'' \mid m_2')) \|c\|_{m_1'}(\Gamma_2^* m_2') \quad \text{(2b)}
\end{align*}
\]

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 \( m_1', m_2' \),

\[
m_2' \otimes (\epsilon) \in \text{cost}(c'')(m_1'(0)) \implies \epsilon \leq \epsilon
\]

Note that off-the-shelf tools can be used to verify that \( c'' \leq \epsilon \) holds for some \( \epsilon \).

**Theorem 1** (Soundness). Given \( c, c', c'', m_1', m_2', \Gamma_1, \Gamma_2, \epsilon \) such that \( \perp \Gamma \Gamma_1[c \rightarrow c'] \Gamma_2 \land c' \Rightarrow c'' \land c'' \leq \epsilon \), one of the following holds:

\[
\begin{align*}
&\max \left\{ \|c''\|_{m_1'}(S) - \exp(\epsilon)\|c\|_{m_1'}(\Gamma_2^* S) \right\} \leq 0, \quad \text{(3a)} \\
&\max \left\{ \|c''\|_{m_1'}(S) - \exp(\epsilon)\|c\|_{m_1'}(\Gamma_2^* S) \right\} \leq 0. \quad \text{(3b)}
\end{align*}
\]

**Proof.** By definition of \( c'' \leq \epsilon \), we have \( \max(c'' \mid m_2') \leq \epsilon \) for all \( m_2' \in S \). Thus, by Lemma 3, we have one of the two:

\[
\|c''\|_{m_1'}(m_2') \leq \exp(\epsilon)\|c\|_{m_1'}(\Gamma_2^* m_2'), \quad \forall m_2' \in S,
\|c''\|_{m_1'}(m_2') \leq \exp(\epsilon)\|c\|_{m_1'}(\Gamma_2^* m_2'), \quad \forall m_2' \in S.
\]

If the first inequality is true, then

\[
\max \left\{ \|c''\|_{m_1'}(S) - \exp(\epsilon)\|c\|_{m_1'}(\Gamma_2^* S) \right\} \leq 0
\]

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

We now prove the main theorem on differential privacy:

**Theorem 2** (Privacy). Given \( \Gamma_1, \Gamma_2, c, c', c'', \epsilon, \epsilon \) such that \( \Gamma_1^* = \Gamma_1^* \land \perp \Gamma_1[c; \text{return } e] \rightarrow (c'; \text{return } e) \Gamma_2 \land c' \Rightarrow c'' \), we have

\[
c'' \leq \epsilon \Rightarrow c \leq \epsilon \text{-differentially private.}
\]
We have implemented ShadowDP into a trans-compiler\(^1\). By the typing rule, we have \(\perp \vdash \Gamma_1[c \rightarrow c']\Gamma_2\). By the soundness theorem (Theorem 1) and the fact that \(\Gamma_1^n = \Gamma_2^n\), we have \([c']_{m|_S}(S) \leq \exp(\epsilon)[e]_{m|_S}(\Gamma_2^n S)\). For clarity, we stress that all sets are over distinct elements (as we have assumed throughout this paper).

By rule (T-Return), \(\Gamma_2 \vdash e : \text{num}_{(0,a)}\) or \(\Gamma_2 \vdash e : \text{bool}\). For any set of values \(V \subseteq [B]\), let \(S'_\gamma = \{m' \in M' | [e]_{m'} \in V\}\) and \(S_V = \{m \in M | [e]_m \in V\}\), then we have \(\Gamma_2 \vdash e : \text{num}_{(0,a)}\):

\[
\begin{align*}
m \in \Gamma_2 S_V &\Rightarrow m = \Gamma_2 m' \text{ for some } m' \in S_V \\
\Rightarrow [e]_m = [e]_{\Gamma_2 m'} = [e]_{m'} \in V \\
\Rightarrow m \in S_V.
\end{align*}
\]

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

\[
\begin{align*}
[P']_{m|_{\gamma}(y)}(V) &= [c']_{m|_{\gamma}(y)}(S'_\gamma) \\
&\leq \exp(\epsilon)[c']_{m|_{\gamma}(y)}(\Gamma_2^n S'_\gamma) \\
&\leq \exp(\epsilon)[e]_{m|_{\gamma}(y)}(S_V) \\
&= \exp(\epsilon)[P]_{m|_{\gamma}(y)}(V).
\end{align*}
\]

Finally, due to Lemma 1, \([P]_m(V) = [P']_{m|_{\gamma}(y)}(V)\). Therefore, by definition of privacy \(c\) is \(\epsilon\)-differentially private. \(\square\)

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 \(\Gamma_1^n = \Gamma_1^n\) (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\(^1\) 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 \((v_e \leq \epsilon)\) before the return command. This assertion specifies the final goal of proving differential privacy. The implementation follows the typing rules explained in Section 4.

Verification The goal of verification is to prove the assertion \((v_e \leq \epsilon)\) never fails for any possible inputs that satisfy the precondition (i.e., the adjacency requirement). To demonstrate the usefulness of the transformed programs, we use a model checker CPAChecker [11] v1.8. CPAChecker is capable of automatically verifying C program with a given configuration. In our implementation, predicate analysis is used. Also, CPAChecker has multiple solver backends such as MathSat [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 \(q[i] + \eta_1\) with the noisy threshold \(\tilde{T}\), and returns the result \((\text{true} \text{ or } \text{false})\). 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 \text{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 \(p\) 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 \(q[i] + \eta_2 \geq \hat{T}\), it outputs the value of the gap \(q[i] + \eta_2 - \hat{T}\) (how much larger the noisy answer is
Table 1. Time spent on type checking and verification

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Type Check (s)</th>
<th>Verification by ShadowDP (s)</th>
<th>Verification by [2] (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Report Noisy Max</td>
<td>0.465</td>
<td>1.932</td>
<td>22</td>
</tr>
<tr>
<td>Sparse Vector Technique (N = 1)</td>
<td>0.398</td>
<td>1.856</td>
<td>27</td>
</tr>
<tr>
<td>Sparse Vector Technique</td>
<td>0.399</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numerical Sparse Vector Technique (N = 1)</td>
<td>0.418</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Numerical Sparse Vector Technique</td>
<td>0.421</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gap Sparse Vector Technique</td>
<td>0.424</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partial Sum</td>
<td>0.445</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prefix Sum</td>
<td>0.449</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Smart Sum</td>
<td>0.603</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rewrite</td>
<td></td>
<td>2.629</td>
<td>580</td>
</tr>
<tr>
<td>Fix ϵ</td>
<td></td>
<td>1.679</td>
<td></td>
</tr>
</tbody>
</table>

function SVT(ε, size, T, N, v, q, i) → list bool
returns (out : list bool)
precondition ∀i ≥ 0. –1 ≤ q[i] ≤ 1 ∧ q[i] ≥ q[0]

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

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

compared to the noisy threshold). Note that the noisy query value q[i] + η₂ is reused for both this check and the output (whereas other proposals either (1) draw fresh noise and result in a larger ε [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 - T)::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 η₂ from Ω ? 2 : 0 to Ω ? (1 - q[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 vₑ := vₑ + (1 - q[i]) × ε/4N; to assert ((1 - q[i]) ≤ 2); vₑ := vₑ + 2 × ε/4N; and provide nonlinear loop invariants; then it is verified (we also verified a variant where ε is fixed to 1).

6.3 Experiments

ShadowDP is evaluated on Report Noisy Max algorithm (Figure 1) along with all the algorithms discussed in Section 6.2, as well as Partial Sum, Prefix Sum and Smart Sum algorithms that are included in the 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 ε-differentially private algorithms with ShadowDP.
6.4 Proof Automation
ShadowDP requires two kinds of annotations: (1) function specification and (2) annotation for sampling commands. As most verification tools, (1) is required since it specifies the property being verified. In all of our verified examples, (2) is fairly simple and easy to write. To further improve the usability of ShadowDP, we discuss some heuristics to automatically generate the annotations for sampling commands. Sampling commands requires two parts of annotation:

1. **Selectors.** The selector has two options: aligned (◦) or shadow (†), with potential dependence. The heuristic is to enumerate branch conditions. For Report Noisy Max, there is only one branch condition Ω, giving us four possibilities: o / † / Ω ? o : † / Ω ? † : o.

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 Ω ? 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 SparseVector. 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 SparseVector 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


Appendix

A ShadowDP Semantics

Given a distribution \( \mu \in \text{Dist}(A) \), its support is defined as \( \text{support}(\mu) = \{ a \mid \mu(a) > 0 \} \). 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 \( \text{unit} \) and \( \text{bind} \) functions to formalize the semantics for commands:

\[
\begin{align*}
\text{unit} & : A \rightarrow \text{Dist}(A) \triangleq \lambda a. \mathbb{1}_a \\
\text{bind} & : \text{Dist}(A) \rightarrow (A \rightarrow \text{Dist}(B)) \rightarrow \text{Dist}(B) \\
& \triangleq \lambda \mu f. (\lambda b. \sum_{a \in A} (f a b) \times \mu(a))
\end{align*}
\]

That is, \( \text{unit} \) takes an element in \( A \) and returns the Dirac distribution where all mass is assigned to \( a \); \( \text{bind} \) 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 rules 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. \( \text{bind}(c, \Gamma)^\dagger \) 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 \( x + n^\dagger \) given \( \Gamma + x : \langle n^\dagger, n^\dagger \rangle \) whenever the shadow value of \( x \) is used; correspondingly,

\[
\langle r, \Gamma \rangle^\dagger = r \quad \langle \text{true}, \Gamma \rangle^\dagger = \text{true} \quad \langle \text{false}, \Gamma \rangle^\dagger = \text{false}
\]

\[
\langle x, \Gamma \rangle^\dagger = \begin{cases} 
  x + n^\dagger & \text{else if } \Gamma - x : \text{num}(\alpha^\dagger, n^\dagger) \\
  x & \text{else} 
\end{cases}
\]

\[
\langle e_1 \text{ op } e_2, \Gamma \rangle^\dagger = \langle e_1, \Gamma \rangle^\dagger \text{ op } \langle e_2, \Gamma \rangle^\dagger \quad \text{where op = } \oplus \lor \otimes \lor \odot
\]

\[
\langle e_1, e_2, \Gamma \rangle^\dagger = \begin{cases} 
  \langle e_1[e_2] + \epsilon_1^\ast[e_2], \text{if } \Gamma + e_1 : \text{list num} \} \\
  \langle e_1[e_2], \Gamma \rangle^\dagger & \text{else if } \Gamma + e_1 : \text{list num}^\ast \\
  \langle e_1[e_2], \Gamma \rangle^\dagger & \text{else} 
\end{cases}
\]

\[
\langle e_1 \text{ ? } e_2, e_3, \Gamma \rangle^\dagger = \langle e_1 \rangle^\dagger ? \langle e_2, \Gamma \rangle^\dagger : \langle e_3, \Gamma \rangle^\dagger
\]

Figure 8. Transformation of numerical expressions for aligned and shadow execution, where \( \ast \in \{ \circ, \dagger \} \).

\[
\langle \text{skip}, \Gamma \rangle^\dagger = \text{skip} \\
\langle c_1 \text{; } e_1 \rangle^\dagger = c_1^\ast \quad \langle c_2 \text{; } e_2 \rangle^\dagger = c_2^\ast \\
\langle c_1 \text{; } c_2 \text{; } e_2 \rangle^\dagger = c_1^\ast ; c_2^\ast \\
\langle x := e, \Gamma \rangle^\dagger = \langle \text{x}^\dagger := (\langle e, \Gamma \rangle^\dagger - x) \\
\langle c_1 \text{; } e_1 \rangle^\dagger = c_1^\ast \quad \text{if } i \in \{1, 2\} \\
\langle \text{if } e \text{ then } c_1 \text{ else } c_2, \Gamma \rangle^\dagger = \langle \text{if } \langle e, \Gamma \rangle^\dagger \text{ then } c_1^\ast \text{ else } c_2^\ast \\
\langle c, \Gamma \rangle^\dagger = c' \\
\langle \text{while } e \text{ do } c, \Gamma \rangle^\dagger = \text{while } \langle e, \Gamma \rangle^\dagger \text{ do } c'
\]

Figure 9. Shadow execution for commands.

we update \( \bar{x}^\dagger \) to \( v - x \) instead of updating the shadow value of \( x \) to some value \( v \).

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 \( \eta_g \) when the noisy query answer is above the noisy threshold, and then releases \( q[i] + \eta_g \) 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 \((0,–)\), which implies that the numerical query \( q[i]+\eta_3 \) should have distance \((0,–)\). We can deduce that \( \eta_3 \) must have distance \( -\tilde{\eta}[i] \) inside the branch, thus we write \( \langle o, -\tilde{\eta}[i] \rangle \) for \( \eta_3 \). The rest of the annotations remain the same as standard Sparse Vector Technique.

```plaintext
∀, count := count + 1;
12 else
14 i := i + 1;
out := 0:: out ;
out := (q[i] +
out := 0:: out ;
out := 0:: out ;
```

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

```plaintext
v_e := \Theta;
2 havoc \eta_1; \ v_e := v_e + \epsilon/3;
3 \ T := T + \eta_1;
4 count := 0; \ i := 0;
5 while (count < N \land i < size)
6 \ assert (count < N \land i < size);
7 \ havoc \ \eta_2; \ v_e := \Omega(\epsilon/3); \ (v_e + 0);
8 if (q[i] + \eta_2 \geq \ T) then
9 \ assert (q[i] + \tilde{\eta}[i] + \eta_2 + 2 \geq \ T + 1);
10 havoc \ \eta_3; \ v_e := v_e + |\tilde{\eta}[i]| \epsilon/3N;
11 out := (q[i] + \eta_3)::out;
12 count := count + 1;
13 else
14 \ assert (\langle q[i] + \tilde{\eta}[i] + \eta_2 \geq \ T + 1));
15 out := 0::out;
16 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

\[ v_e = v_e + |\tilde{\eta}[i]| \epsilon/3; \]

\[ \text{to assert } (|\tilde{\eta}[i]| \leq 1); \ v_e = v_e + \epsilon/3; \]

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

C.2 Partial Sum

We now study an \( \epsilon \)-differentially private algorithm Partial-Sum (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 \((0,–)\), it is easy to deduce that the distance of \( \eta \) should be \(-\sum \). Adding the annotation \( \langle o, -\sum \rangle \) to Line 5 in Figure 11, ShadowDP successfully transforms and verifies the program. Note that since the cost update command \( v_e := v_e + |\sum| \times \epsilon; \) contains non-linear arithmetic, we carefully rewrite this command to assert \( (|\sum| \leq 1); \ v_e := v_e + \epsilon; \). ShadowDP is able to type check and verify this algorithm within seconds.

```plaintext
function PartialSum (\epsilon, size : num(0,0) ; q : list num(\epsilon, \epsilon))
returns (out : num(0,0))
 precondition \( \forall i \geq 0. \ -1 \leq \tilde{q}[i] \leq 1 \land \tilde{q}[i] = \tilde{\eta}[i] \)
```

```
sum := 0; i := 0;
2 while (i < size)
3 \ sum := sum + q[i];
4 \ i := i + 1;
5 \ \eta := Lap (1/\epsilon); \ o, -\sum;
6 \ out := sum + \eta;
```

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

```plaintext
sum := 0; i := 0;
2 \ sum := 0;
3 \ while (i < size)
4 \ \assert (i < size);
5 \ sum := sum + q[i];
6 \ sum := sum + \tilde{\eta}[i];
7 \ i := i + 1;
8 \ havoc \ \eta; \ v_e := v_e + |\sum| \times \epsilon;
9 \ out := sum + \eta;
```

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.

```plaintext
function SmartSum(ε, M, T : list num(0,)) q : list num(s,))
returns (out : list num(0,))
precondition M ≥ 0. 0 ≤ i ≤ T = 1 ∧ (∃j : i, q[j] ≥ 0 ⇒ (∀i, q[i] = 0))
1 next := 0; i := 0; sum := 0;
2 while i ≤ T
3     if (i + 1) mod M = 0 then
4         m := Lap (1/ε), o, i, −sum − q[i];
5         next := sum + q[i] + m;
6         sum := 0;
7         out := next; out;
8     else
9         m := Lap (1/ε), o, −q[i];
10        next := sum + q[i] + m;
11        sum := 0;
12        out := next; out;
13    i := i + 1;
14
The transformed program:
1 next := 0; n := 0; i := 0; sum := 0;
2 sum := 0;
3 while (i < size ∧ i ≤ T) assert (i < size ∧ i ≤ T);
4 if (i + 1) mod M = 0 then
5     havoc m; v := v + |−sum − q[i]| × ε;
6     next := sum + q[i] + m;
7     sum := 0;
8     out := next; out;
9     sum := 0;
10 else
11     havoc m; v := v + |−q[i]| × ε;
12     next := sum + q[i] + m;
13     sum := 0;
14     out := next; out;
15     sum := 0;
16    i := i + 1;
17
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 −sum − q[i] to η1 and −q[i] to η2, and use o 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

\[ v + |−sum − q[i]| × \epsilon; \]
\[ v := v + |−q[i]| × \epsilon; \]

to

\[ if (|−sum − q[i]| > 0) \]
\[ assert (|−sum − q[i]| ≤ 1); v := v + \epsilon; \]
\[ if (|−q[i]| > 0) \]
\[ assert (|−q[i]| ≤ 1); v := v + \epsilon; \]

Moreover, one difference of this algorithm is that it satisfies 2-ε-differential privacy [14] instead of ε-differential privacy, thus the last assertion added to the program is changed to assert (v ≤ 2 × ε). With this CPAchecker is able to verify this algorithm.

We also verified a variant of SmartSum, called Prefix Sum algorithm [2]. This algorithm is a simplified and less precise version of SmartSum, 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 η2, type checking and transformation follows SmartSum. Note that Prefix Sum satisfies ε-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 v. Moreover, we assume that memory tracks the entire list of sampled random values at each point.

**Lemma 4 (Numerical Expression).** \( v, m', \Gamma, \text{ such that } \Gamma \vdash e : \text{num}(s, s'), \) we have

\[ [e]_{m'} + [\epsilon]_{s'} = [v]_{\Gamma m'}; \]
\[ [e]_{m'} + [\epsilon]_{s'} = [v]_{\Gamma m'}. \]

**Proof.** Induction on the inference rules.

- (T-Num): trivial.
- (T-Var): If \( \Gamma'(x) = \star, \) is \( \Gamma\). By Definition 3, \( \Gamma m'(x) = m'(x) + m'(\tau)\). If \( \Gamma'(x) = \emptyset, \) is \( \emptyset\). Again, we have \( \Gamma m'(x) = m'(x) + [\emptyset]_{s'} \) by Definition 3. The case for \( \Gamma\) is similar.
- (T-Plus, T-Times, T-Ternary, T-Index): by the induction hypothesis (list is treated as a collection of variables of the same type).

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

\[ [e]_{m'} = [v]_{\Gamma m'} = [e]_{\Gamma m'}. \]

\[ \square \]
Proof. Induction on the inference rules.
• (T-Boolean): trivial.
• (T-Var): the special case where \( \phi = \psi = 0 \). Result is true by Definition 3.
• (T-ODot): by Lemma 4 and induction hypothesis, we have \( \llbracket e_1 \rrbracket_{m'} + \llbracket e_2 \rrbracket_{m'} = \llbracket e_1 \rrbracket_{\Gamma} \rrbracket_{m'} \) for \( i \in \{1, 2\} \). By the assumption of (T-ODot), we have
\[
\llbracket e_1 \odot e_2 \rrbracket_{m'} \iff \llbracket (e_1 + e_2) \odot (e_1 + e_2) \rrbracket_{m'} = \llbracket e_1 \odot e_2 \rrbracket_{\Gamma} \rrbracket_{m'}.
\]
The case for \( \Gamma \) is similar.
• (T-Ternary, T-Index): by the induction hypothesis.

D.1 Injectivity

We first prove that the type system maintains injectivity.

Lemma 6. For all \( c', pc, m', m'_1, m'_2, \Gamma_1, \Gamma_2, pc' \models \Gamma_1; \Gamma_2 \), there exists some \( m'_3 \) and \( m'_4 \) such that
\[
\llbracket c' \rrbracket_{\Gamma}(m'_3) \neq 0 \land \llbracket c' \rrbracket_{\Gamma}(m'_4) \neq 0 \land \llbracket e' \rrbracket_{m'}(m'_3) \neq 0 \land \llbracket e' \rrbracket_{m'}(m'_4) \neq 0.
\]

Proof.

By structural induction on \( c \).

• Case \( \phi \): trivial since it is non-probabilistic.
• Case \( \phi \equiv \epsilon \): trivial since it is non-probabilistic.
• Case \( \phi \equiv \epsilon \phi \): Let \( pc' \models \Gamma_1 \{ \epsilon \rightarrow c' \} \Gamma_2 \). Then exists some \( m'_3 \) and \( m'_4 \) such that
\[
\llbracket c' \rrbracket_{\Gamma}(m'_3) \neq 0 \land \llbracket c' \rrbracket_{\Gamma}(m'_4) \neq 0 \land \llbracket e' \rrbracket_{m'}(m'_3) \neq 0 \land \llbracket e' \rrbracket_{m'}(m'_4) \neq 0.
\]

If \( m'_3 = m'_4 \), results is true by the induction hypothesis on \( c' \). Otherwise, we have \( \exists \eta. \llbracket ? \rrbracket_{\Gamma}(\eta) = \llbracket c' \rrbracket_{\Gamma}(m'_3) \neq 0 \land \llbracket c' \rrbracket_{\Gamma}(\eta) = \llbracket c' \rrbracket_{\Gamma}(m'_4) \neq 0 \). Hence, \( \llbracket c' \rrbracket_{\Gamma}(m'_3) = \llbracket c' \rrbracket_{\Gamma}(m'_4) \) for any \( \eta \). Otherwise, we have \( \exists \eta. \llbracket ? \rrbracket_{\Gamma}(\eta) = \llbracket c' \rrbracket_{\Gamma}(m'_3) \neq 0 \land \llbracket c' \rrbracket_{\Gamma}(\eta) = \llbracket c' \rrbracket_{\Gamma}(m'_4) \neq 0 \). There are two cases:

1. \( m'_3 = m'_4 \): in this case, \( c'_4 \) only changes the value of \( x' \) such that \( \Gamma_1 \models x' \neq x' \). Moreover, we have \( m'_1(x') = \llbracket x' \rrbracket_{m'_1} = \llbracket x' \rrbracket_{m'_2} \neq m'_2(x) \). Hence, \( m'_1 \neq m'_2 \).

2. \( m'_3 \neq m'_4 \): by the induction hypothesis, \( \exists \eta. \llbracket ? \rrbracket_{\Gamma}(\eta) = \llbracket c' \rrbracket_{\Gamma}(m'_3) \neq 0 \land \llbracket c' \rrbracket_{\Gamma}(\eta) = \llbracket c' \rrbracket_{\Gamma}(m'_4) \neq 0 \). When \( pc = \top \), \( c' \) also includes \( e' \). In this case, result still holds after \( c' \) since \( c' \) is deterministic by construction.

Case \( \phi \odot \phi \): let \( pc' \models \Gamma_1 \{ \phi \rightarrow c' \} \Gamma_2 \). We proceed by induction on the number of iterations:

1. \( c \) is not executed: \( m'_1 = m'_2 \).
2. \( c \) is executed \( n + 1 \) times: similar to the \( c_1; c_2 \) case.

When \( pc = \top \), \( c' \) also includes \( c' \). In this case, result still holds after \( c' \) since \( c' \) is deterministic by construction.

Proof of Lemma 2


D.2 Instrumentation

Next, we show a property offered by the auxiliary function \( \Gamma_1, \Gamma_2, pc \Rightarrow c' \) 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). For all \( pc, \Gamma_1, \Gamma_2, c' \), if \( (\Gamma_1, \Gamma_2, pc) \Rightarrow c' \), then for any memory \( m'_1 \), there is a unique \( m'_2 \) such that
\[
\llbracket c' \rrbracket_{\Gamma_1}(m'_1) \neq 0 \land \llbracket c' \rrbracket_{\Gamma_2}(m'_2) \neq 0 \land \llbracket c' \rrbracket_{\Gamma_1}(m'_1) = \llbracket c' \rrbracket_{\Gamma_2}(m'_2).
\]

Proof.

By construction, \( c' \) is deterministic. Hence, there is a unique \( m'_2 \) such that \( \llbracket e' \rrbracket_{m'_2} = \llbracket c' \rrbracket_{\Gamma_1}(m'_1) \neq 0 \).

Consider any variable \( x \in Vars \). By the construction of \( c' \), we note that \( m'_1(x) = m'_2(x) \) and \( m'_2(x) \) differs in \( m'_1 \) and \( m'_2 \) only if \( \Gamma_1 \models x \neq x \) and \( \Gamma_2 \models x \neq x \). In this case, \( \Gamma_1 \models m'_1(x) = m'_2(x) + \llbracket x \rrbracket_{m'_1} = m'_2(x) + \llbracket x \rrbracket_{m'_2} \). Otherwise, \( \Gamma_1 \models m'_1(x) = m'_2(x) \) for some \( x \). In this case, \( \Gamma_1 \models m'_1(x) = m'_2(x) \) holds. When \( \Gamma_1 \models x \neq x \), \( \Gamma_2 \models x = x \). Hence, \( \Gamma_1 \models x \neq x \) for some \( x \). In this case, \( \Gamma_1 \models m'_1(x) = m'_2(x) \).

When \( pc = \top \), the same argument applies to the case of \( \Gamma_1 \models \Gamma_1 \models \Gamma_1 \).

D.3 Shadow Execution

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

Lemma 8. For all \( \phi, \Gamma, m', \phi' \), if \( e \) is well-typed under \( \Gamma \), we have
\[
\llbracket (e, \Gamma) \rrbracket_{m'} = \llbracket e \rrbracket_{\Gamma} \rrbracket_{m'}.
\]

Proof.

By structural induction on the \( e \):

- \( e \) is \( \phi \): trivial.
- \( e \) is \( \phi \) if \( \phi \): with base type of type \( m \), \( \llbracket (e, \Gamma) \rrbracket_{m'} = m' = m + \llbracket e \rrbracket_{\Gamma} \rrbracket_{m'} \). When \( (x, \Gamma) = \text{num}(\mathbf{r}) \), we have \( \llbracket x \rrbracket_{m'} = m + \llbracket e \rrbracket_{\Gamma} \rrbracket_{m'} \). Otherwise, \( \llbracket x \rrbracket_{m'} = m' = m' \rrbracket_{m'} \).
- \( e \) is \( \phi \odot \phi \): by the induction hypothesis.
- \( e \) is \( \phi \odot \phi \): By the typing rule (T-Index), \( \Gamma + e \phi \) is well-typed by Lemma 4. Let \( \Gamma' \models e \phi \). Then \( \llbracket e \phi \rrbracket_{m'} = \llbracket e \phi \rrbracket_{m'} \). When \( d = * \) and \( e \phi \neq \text{num}(\mathbf{r}) \), \( m \neq m' \neq m' \).
as \([\langle e_1, \Gamma \rangle \uparrow [e_2]_{m^*}]_{m'}\). By induction hypothesis, this is the same as \([e_1]_{m'} \cdot [e_2]_{m^*} = [e_1]_{\Gamma \downarrow m'}\).

When \(d = \nu^1\), \([\langle e_1 [e_2], \Gamma \rangle \uparrow [e_2]_{m^*}]_{m'}\) is defined as \([e_1]_{\Gamma \downarrow m'} + \nu^1\) \(e_2\), which is the same as \([e_1]_{\Gamma \downarrow m'}\) (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 \([\langle e_1, \Gamma \rangle \uparrow [e_2]_{m^*}]_{m'} = [e_1]_{\Gamma \downarrow m'}\). Hence, the same element is selected on both ends. Result is true by induction hypothesis.

\(\square\)

Next, we show that shadow execution simulates two executions on \(\Gamma^1\)-related memories via two program executions.

Lemma 9 (Shadow Execution). \(\forall e, c^1. \Gamma. (\forall x \in \text{Asgndx}(c). \Gamma^1(x) = x) \land (e, \Gamma)^\uparrow = c^1\), we have

\[\forall m'_1, m'_2, \exists [c^1]_{m'_1}(m'_2) = [e]_{\Gamma^1 \downarrow m'_1}.\]

**Proof.** By structural induction on \(e\). First, we note that the construction of shadow execution does not apply to sampling instructions. Hence, \(c^1\) and \(c\) 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 \(c^1\). Then this lemma can be stated as

\[m'_2 = [c^1]_{m'_1} \Rightarrow \Gamma^1 m'_2 = [e]_{\Gamma^1 \downarrow m'_1}.\]

- \(e\) is skip: we have \(m'_1 = m'_2\) in this case. Hence, \(\Gamma^1 m'_2 = \Gamma^1 m'_1 = [\text{skip}]_{\Gamma \downarrow m'_1}\).
- \(e\) is \((c_1, c_2)\): let \((c_1, \Gamma)^\uparrow = c_1^1\) and \((c_2, \Gamma)^\uparrow = c_2^1\) and \(m' = [c_1^1]_{m'_1}\). By induction hypothesis, we have \(\Gamma^1 m' = [c_1^1]_{\Gamma \downarrow m'_1}\).
- \(e\) is \((x := e)\): we have \(c^1 = (\overline{x} := (e, \Gamma)^\uparrow) = x\) in this case. Moreover, \(m'_2 = m'_1 \cdot [\langle e, \Gamma^1 \rangle \uparrow - x]_{m'_1}(\overline{x})\). By Lemma 8, we have \([\langle e, \Gamma^1 \rangle \uparrow]_{m'_1} = [\text{ skip}]_{\Gamma^1 \downarrow m'_1}\).

For variable \(x\), we know that \(\Gamma(x)^\uparrow = x\) by assumption. So \(\Gamma^1 m'_2(x) = m'_1(x) + m'_2(\overline{x}) = m'_1(x) + [\langle e, \Gamma^1 \rangle \uparrow - x]_{m'_1}(x) = [\langle e, \Gamma^1 \rangle \uparrow]_{m'_1}(x)\). For a variable \(y\) other than \(x\), the result is true since both \(c\) and \(c^1\) do not modify \(y\) and \(y^1\), and \(y^1\)’s distance changes due to the well-formedness check in typing rule (T-Asgn).

- \(e\) is \(\text{if} e\), then \(c_1\) else \(c_2\): let \((c_1, \Gamma)^\uparrow = c_1^1\) and \((c_2, \Gamma)^\uparrow = c_2^1\). In this case, \(c^1 = \text{ if } (e, \Gamma)^\uparrow \text{ then } c_1^1 \text{ else } c_2^1\). By Lemma 8, we have \([\langle e, \Gamma^1 \rangle \uparrow]_{m'_1} = [\text{ skip}]_{\Gamma^1 \downarrow m'_1}\). Hence, both \(c\) and \(c^1\) will take the same branch under \(\Gamma^1 m'_1\) and \(m'_2\) respectively. The desired results follow from induction hypothesis on \(c_1\) or \(c_2\).
- \(e\) is \(e\) where \(e\) do: again, since \([\langle e, \Gamma^1 \rangle \uparrow]_{m'_1} = [\text{ skip}]_{\Gamma^1 \downarrow m'_1}\). The desired result follows from induction hypothesis on the number of loop iterations.

\(\square\)

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). \(\forall e, c^1, c^2. \Gamma. \Gamma \uparrow \Gamma \Gamma_1(c \rightarrow c) \Gamma_2\), then we have

1. \(((\forall x \in \text{Asgndx}(c). \Gamma^1(x) = x) \land (e, \Gamma)^\uparrow = c^1)\)
2. \(\forall m'_1, m'_2, [c^1]_{m'_1}(m'_2) \neq 0 \Rightarrow \Gamma^1 m'_1 = \Gamma^1 m'_2\)

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

- \(c\) is \(\text{skip}\): trivial.
- \(c\) is \(\text{if}\ then \(c_1\) else \(c_2\): when \(pc = \top\), we have \(c' = if\ then\ (assert(\nu^0); c_1; c_1')\) else\ (assert(\nu^0); c_2; c_2').\)

By induction hypothesis, we know that \(c_1'\) and \(c_2'\) does not modify any shadow variable and their ending typing environments, say \(\Gamma_1, \Gamma_2^1\), satisfy condition 1. Hence, the ending environment \(\Gamma_1 \Gamma_2\) satisfies condition 1 too.

To show (2), we assume \(c_1\) is executed without losing generality. Let \(m'_2\) be the memory state between \(c_1'\) and \(c_2'\). By induction hypothesis, \(\Gamma^1 m'_1 = (\Gamma^1)^\uparrow m'_1\). By the definition of \(\Gamma_1 \Gamma_2\), \(\Gamma_1 \Gamma_2\) satisfies condition 1 holds. Moreover, by the definition of \(\Gamma^1 \Gamma_1 \Gamma_2\), \(\Gamma \Rightarrow c_1'\) and \(c_2'\) only modifies \(\overline{x'}, x' \in \text{Vars}\). Hence, \(\Gamma^1 m'_2 = (\Gamma^1)^\uparrow m'_2\).

- \(c\) is \(\text{while}\ e\ do\ c':\) by the definition of \(\mathcal{U}\) and induction hypothesis, condition 1 holds.

\(\square\)

**Lemma 11**. \(\forall e, c^1, c^2, \Gamma_1, \Gamma_2. (\forall x \in \text{Asgndx}(c). \Gamma_1^1(x) = x) \land (c, \Gamma_1)^\uparrow = c^1 \land c' \text{ is deterministic, and } (\forall m'_1, m'_2, [c^1]_{m'_1}(m'_2) \neq 0 \Rightarrow \Gamma^1 m'_1 = \Gamma^1 m'_2)\), then we have

\[\forall m'_1, m'_2, [c^1; c']_{m'_1}(m'_2) = [c]_{\Gamma^1 \downarrow m'_1} m'_2\]

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

\[\forall m'_1, m'_2, [c^1; c']_{m'_1}(m'_2) = [c]_{\Gamma^1 \downarrow m'_1(m'_2)}\]

Let \(m'_1 = [c^1]_{m'_1}\) and \(m'_2 = [c^1]_{m'_2}\). By assumption, we have \(\Gamma^1 m'_1 = \Gamma^1 m'_2\). Moreover, by Lemma 9, we have \(\Gamma^1 m'_2 = [c]_{\Gamma^1 \downarrow m'_2}\). Hence, \(\Gamma^1 m'_2 = [c]_{\Gamma^1 \downarrow m'_2}\).
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 $\|c\|_m'(m'_2) \neq 0$. Hence, in the proof we assume that $\|c\|_m'(m'_2) > 0$.

- Case (skip): trivial.
- Case ($x := e$): By (T-Asgn), we have $\max(c' \mid m'_2) = 0$. This command is deterministic in the sense that when $\|c\|_m'(m'_2) \neq 0$, we have $\|c\|_m'(m'_2) = 1$ and $\perp = m'_2([\|c\|_m'/x])$. To prove (1) and (2a), it suffices to show that $\Gamma^1_2m'_2 = \Gamma^0_1m'_1([\|c\|_m'/x])$ and $\Gamma^2_2m'_2 = \Gamma^0_1m'_1([\|c\|_m'/x])$. We prove the latter one as the other (given $pc = \bot$) can be shown by a similar argument.

First, we show $\Gamma^1_2m'_2(x) = \Gamma^0_1m'_1([\|c\|_m'/x])(x)$. Let $\Gamma^1_1 + e : (v^o, v^1)$. By the typing rule, we have $\Gamma^2_2(x) = v^o, v^1$, and $\Gamma^2_2m'_2(x) = m'_2(x) + [v^o, v^1]m'_1$

- Case ($c_1; c_2$): For any $m'_2$ such that $\|c_1; c_2\|_m'(m'_2) \neq 0$, there exists some $m'$ such that $\|c_1; c_2\|_m'(m') \neq 0$. Let $pc \vdash \Gamma_1 \vdash c_1 \vdash c_1'[\Gamma], pc \vdash \Gamma_2 \vdash c_2 \vdash c_2'[\Gamma].$ For (1), we have

$$\|c_1; c_2\|_m'(m'_2) = \sum_{m'} \|c_1; c_2\|_m'(m' \cdot c_2)[\Gamma] \leq \sum_{m'} \|c_1; c_2\|_m'(m' \cdot c_2)[\Gamma] \leq \sum_{m'} \|c_1; c_2\|_m'(m' \cdot c_2)[\Gamma] \leq \|c_1; c_2\|_m'(m'_2).$$

The second inequality is due to well-formedness, and the forth equality is due to Lemma 4.

Second, we show that $\Gamma^2_2m'_2(y) = \Gamma^0_1m'_1([\|c\|_m'/x])$ for $y \neq x$. First, by Rule (T-Asgn), $\Gamma^2_2(y) = \Gamma^2_1(y), m'_1(y) = m'_1(y), m'_2(y) = m'_2(y), m'_2(y) = m'_2(y)$. If $\Gamma^2_2(y) = \Gamma^2_1(y) = v, \Gamma^2_2m'_2(y) = m'_2(y) + [v, v]m'_1(x) = \Gamma^2_1m'_1(x) \neq \Gamma^2_1m'_1(x)$, where $\|v\|_m = \|v\|_m$ due to well-formedness.

- Case ($c_1; c_2$): For any $m'_2$ such that $\|c_1; c_2\|_m'(m'_2) \neq 0$, there exists some $m'$ such that $\|c_1; c_2\|_m'(m') \neq 0$. Let $pc \vdash \Gamma_1 \vdash c_1 \vdash c_1'[\Gamma], pc \vdash \Gamma_2 \vdash c_2 \vdash c_2'[\Gamma].$ For (1), we have

$$\|c_1; c_2\|_m'(m'_2) = \sum_{m'} \|c_1; c_2\|_m'(m' \cdot c_2)[\Gamma] \leq \sum_{m'} \|c_1; c_2\|_m'(m' \cdot c_2)[\Gamma] \leq \sum_{m'} \|c_1; c_2\|_m'(m' \cdot c_2)[\Gamma] \leq \|c_1; c_2\|_m'(m'_2).$$

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 $\epsilon_1 = \max(c'_1 \mid m'_1), \epsilon_2 = \max(c'_2 \mid m'_2)$ and $\epsilon = \max(c'_1; c'_2 \mid m'_1; m'_2)$. Note that $\epsilon_1 + \epsilon_2 \leq \epsilon$ due to the fact that $m'_2 \cup (\epsilon_1 + \epsilon_2) \in [c'_1; c'_2 \mid m'_1; m'_2]$. By induction hypothesis, we have one of the following two cases.

1. $\|c'_1; c'_2\|_m'(m'_2) \leq \exp(\epsilon_2)[c'_2][\Gamma^1_2m'_2].$ We have

$$\|c'_1; c'_2\|_m'(m'_2) = \sum_{m'} \|c'_1; c'_2\|_m'(m' \cdot c_2)[\Gamma] \leq \exp(\epsilon_2) \sum_{m'} \|c'_1; c'_2\|_m'(m' \cdot c_2)[\Gamma^1_2m'_2] \leq \exp(\epsilon_2) \sum_{m'} \|c'_1; c'_2\|_m'(m' \cdot c_2)[\Gamma^1_2m'_2] \leq \exp(\epsilon_2)[c'_1; c'_2][\Gamma^1_2m'_2].$$

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. $\|c'_1; c'_2\|_m'(m'_2) \leq \exp(\epsilon_1)[c'_1][\Gamma^1_2m'_2].$ By induction hypothesis on $c_1$, we have two more cases:
   a. $\|c'_1\|_m'(m') \leq \exp(\epsilon_1)[c'_1]_{m'}(m'').$ In this case,

$$\|c'_1; c'_2\|_m'(m'_2) \leq \sum_{m'} \|c'_1; c'_2\|_m'(m' \cdot c_2)[\Gamma] \leq \exp(\epsilon_1) \sum_{m'} \|c'_1; c'_2\|_m'(m' \cdot c_2)[\Gamma^1_2m'_2] \leq \exp(\epsilon_1)[c'_1; c'_2][\Gamma^1_2m'_2].$$

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. $\|c'_1\|_m'(m') \leq \exp(\epsilon_1)[c'_1]_{m'}(m'').$ We have

$$\|c'_1; c'_2\|_m'(m'_2) \leq \exp(\epsilon)[c'_1; c'_2][\Gamma^1_2m'_2].$$

by a similar argument as above.

- Case (if $e$ then $c_1$ else $c_2$): If $\Gamma_1 + e : \text{bool}$, then by Lemma 5 we have $\|e\|_m = \|e\|_m \neq \|e\|_m$. Hence, the same branch is taken in all related executions. By rule (T-Ir), the transformed program is

$$\text{if } e \text{ then } (\text{assert } ((\epsilon_2)'; c'_1; c'_2') \text{ else } (\text{assert } (\neg(\epsilon_2)'); c'_1; c'_2')).$$

and $\Gamma_1, \Gamma_1 \cup \Gamma_2, \perp \Rightarrow c'_1, c_2, \Gamma_1 \cup \Gamma_2, \perp \Rightarrow c'_2$. Without loss of generality, suppose that $c_1$ is executed in all related executions. By Lemma 7, there is a unique $m'$ such that $\|c'_1\|_m'(m') \neq 0$. By induction hypothesis, we have

$$\|c'_1\|_m'(m') \leq \|c'_1\|_m'(m')$$

Moreover, by Lemma 7, $\Gamma^1_1 m'_2 = \Gamma^1_2 m'_2$ and

$$\|c'_1; c'_1\|_m'(m'_2) = \|c'_1\|_m'(m').$$

Hence, we have

$$\|c'_1; c'_1\|_m'(m'_2) \leq \|c'_1; c'_2\|_m'(m' \cdot c_2)[\Gamma] \leq \exp(\epsilon)[c'_1; c'_2][\Gamma^1_2m'_2].$$

(2a) or (2b) can be proved in a similar way.
When $\Gamma_1 \not\vdash e : \mathsf{bool}$, 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

$$\text{if} \ e \ \text{then} \ \langle (\langle e \rangle^\circ); c; c'_1 \rangle \ \text{else} \ \langle (\neg (\langle e \rangle^\circ)); c; c''_1 \rangle$$

\[ \text{if} \ e \ \text{then} \ c_1 \ \text{else} \ c_2, \Gamma_1 \cup \Gamma_2 \uparrow \]

By Lemma 10 and the definition of $\Rightarrow$ under high pc, we have that $\forall m_1', m_2'. \ (\|c\| m_1'(m_2') \neq 0 \Rightarrow \Gamma_1^1 m_1' = \Gamma_2^1 m_1'$, and $\Gamma_1 \cup \Gamma_2(x) = \forall x \in \mathsf{Asgn}(c_1; c_2)$. Furthermore, the program is deterministic since it type-checks under $\top$. Therefore, $\max(W'' \ | m_1') = 0$ and (1) holds by Lemma 11.

• Case (while $e$ do $c$): Let $W' = \text{while} \ e \ \text{do} \ c$. If $\Gamma_1 \vdash e : \mathsf{bool}$, then $\|e\| m_1' = \|e\| c_1', \Gamma_1 \cup \Gamma_2 \uparrow$. By rule (T-While), the transformed program is

$$W' = c_1; \text{while} \ e \ \text{do} \ (\langle (\langle e \rangle^\circ); c'; c'' \rangle)$$

where $\top \vdash \Gamma_2 \cup \Gamma_2 \ (e \rightarrow c') \ \Gamma_2, (\Gamma_2, \Gamma_1 \cup \Gamma_2, \top) \Rightarrow c''$. We proceed by natural induction on the number of loop iterations (denoted by $i$).

When $i = 0$, we have $\|e\| m_1' = \text{false}$. The semantics, we have $\|W'\| m_1' = \text{unit}(\|c_1\| \Gamma_1^1 m_1')$, $\|W\| \Gamma_1^1 m_1' = \text{unit}(\|c_2\| \Gamma_2^1 m_1')$ and $\|W\| \Gamma_2^1 m_1' = \text{unit}(\|c_2\| \Gamma_2^1 m_1')$. Furthermore, $\max(W'' \ | m_1') = 0$. By Lemma 7, $(\Gamma_2^1 m_1') = (\Gamma_2^1 (\|c_2\| \Gamma_1^1 m_1'))$ and $(\Gamma_2^1 m_1') = (\Gamma_2^1 (\|c_2\| \Gamma_1^1 m_1'))$. So desired result holds.

When $i = j + 1 > 0$, we have $\|e\| m_1' = \text{true}$. By the semantics, we have $\|W'\| m_1' = \text{true}(\|c_1\| \Gamma_1^1 m_1')$ and $\|W\| \Gamma_1^1 m_1' = \text{true}(\|c_1\| \Gamma_1^1 m_1')$. Furthermore, $\max(W'' \ | m_1') = 0$. By Lemma 7, $(\Gamma_2^1 m_1') = (\Gamma_2^1 (\|c_2\| \Gamma_1^1 m_1'))$ and $(\Gamma_2^1 m_1') = (\Gamma_2^1 (\|c_2\| \Gamma_1^1 m_1'))$. So desired result holds.

In the second equation, we have $m_1'(x) = m_1'(x)$ because $m_1 = m_1'(x)$ and $x \neq \eta$. The third equation is due to the well-formedness assumption. Also, $\Gamma_2^1 m_1'(\eta) = \eta$ since $\Gamma_2^1 (\eta) = \eta$. Therefore, we have $\Gamma_2^1 m_1' = \Gamma_1^1 m_1'(v \eta)$ and thus $\|c\| \Gamma_1^1 m_1'(\Gamma_2^1 m_1') = \mu_\eta(v) = \mu_\eta(c_2).$

Similarly, we can show that $\Gamma_2^1 m_1' = \Gamma_1^1 m_1'(v \eta)$ and therefore $\|c\| \Gamma_1^1 m_1'(\Gamma_2^1 m_1') = \mu_\eta(v) = \mu_\eta(c_2).$

Furthermore, since $\nu_\eta := \nu_\eta + |\nu_\eta|/r$, the set $\{c'' \ | m_1'\}$ contains a single element $\|\nu_\eta|/r\| m_1' = |d|/r$, and thus max$c''(\ | m_1') = |d|/r$. Therefore, we have

$$\|c''\| m_1'(\Gamma_2^1 m_1') = \mu_\eta(v) \leq \exp(|d|/r)\mu_\eta(v + d)$$

The second inequality is due to (4). Thus (1) and (2a) hold. Next consider the case $S = \top$. By the typing rule, we have $\Gamma_1^1 (x) = \Gamma_2^1 (x)$ for $x \neq \eta$, and $\Gamma_1^1 (\eta) = \langle \nu_\eta, 0 \rangle$. By a similar argument we have $\Gamma_2^1 m_1' = \Gamma_1^1 m_1'(v \eta)$ and $\Gamma_2^1 m_1' = \Gamma_1^1 m_1'(v + d \eta)$ and $\nu_\eta := \nu_\eta + |\nu_\eta|/r$, the set $\{c'' \ | m_1'\}$ contains a single element $\|\nu_\eta|/r\| m_1' = |d|/r$, and thus max$c''(\ | m_1') = |d|/r$. Therefore, we have

$$\|c''\| m_1'(\Gamma_2^1 m_1') = \mu_\eta(v) = \exp(|d|/r)\mu_\eta(v + d)$$

Thus (1) and (2b) hold. Finally, consider $S = e ? S_1 : S_2$. For any $m_1'$, we have $S = S_1$ if $\|e\| m_1' = \text{true}$ and $S = S_2$ if $\|e\| m_1' = \text{false}$.
induction, $S$ evaluates to either $\circ$ or $\dagger$ under $m'$. 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 $\llbracket c \rrbracket : M \rightarrow \mathcal{P}(M)$. The semantics of commands are formalized as follows.

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