Solo: A Lightweight Static Analysis for Differential Privacy

CHIKÉ ABUAH, University of Vermont, USA
DAVID DARAIS, Galois, Inc., USA
JOSEPH P. NEAR, University of Vermont, USA

Existing approaches for statically enforcing differential privacy in higher order languages use either linear or relational refinement types. A barrier to adoption for these approaches is the lack of support for expressing these "fancy types" in mainstream programming languages. For example, no mainstream language supports relational refinement types, and although Rust and modern versions of Haskell both employ some linear typing techniques, they are inadequate for embedding enforcement of differential privacy, which requires "full" linear types. We propose a new type system that enforces differential privacy, avoids the use of linear and relational refinement types, and can be easily embedded in richly typed programming languages like Haskell. We demonstrate such an embedding in Haskell, demonstrate its expressiveness on case studies, and prove soundness of our type-based enforcement of differential privacy.

CCS Concepts: • **Security and privacy → Logic and verification**; • **Theory of computation → Logic and verification**.

Additional Key Words and Phrases: Differential privacy, verification, typechecking

ACM Reference Format:
Chiké Abuah, David Darais, and Joseph P. Near. 2022. Solo: A Lightweight Static Analysis for Differential Privacy. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 150 (October 2022), 30 pages. https://doi.org/10.1145/3563313

1 INTRODUCTION

Differential privacy has become the standard for protecting the privacy of individuals with formal guarantees of *plausible deniability*. It has been adopted for use at several high-profile institutions such as Google [Úlfar Erlingsson et al. 2014], Facebook [Nayak 2020], and the US Census Bureau [Abowd 2018]. However, experience has shown that implementation mistakes in differentially private algorithms are easy to make and difficult to catch [Lyu et al. 2017]. Verifying that differentially private programs *actually* ensure differential privacy is thus an important problem, given the sensitive nature of the data processed by these programs.

Recent work has made significant progress towards techniques for static verification of differentially private programs. Existing techniques typically define novel programming languages that incorporate specialized static type systems (linear types [Near et al. 2019; Reed and Pierce 2010], relational types [Barthe et al. 2015], dependent types [Gaboardi et al. 2013], etc.). However, there remains a major challenge in bringing these techniques to practice: the specialized features they rely on do not exist in mainstream programming languages.

We introduce Solo, a novel type system for static verification of differential privacy that does *not* rely on linear types, implemented in Haskell. Solo is similar to Fuzz [Reed and Pierce 2010]

Authors' addresses: Chiké Abuah, cabuah@uvm.edu, University of Vermont, USA; David Darais, darais@galois.com, Galois, Inc., USA; Joseph P. Near, jnear@uvm.edu, University of Vermont, USA.

2475-1421/2022/10-ART150
https://doi.org/10.1145/3563313

and its descendants in expressive power, but Solo can be implemented entirely in Haskell with no additional language extensions. In particular, Solo's sensitivity and privacy tracking mechanisms are compatible with higher-order functions, and leverage Haskell's type inference system to minimize the need for additional type annotations.

In differential privacy, the *sensitivity* of a computation determines how much noise must be added to its result to achieve differential privacy. Fuzz-like languages track sensitivity relative to program variables, using a linear typing discipline. The key innovation in Solo is to track sensitivity relative to a set of global *data sources* instead, which eliminates the need for linear types. Compared to prior work on static verification of differential privacy, our system can be embedded in existing programming languages without support for linear types, and supports advanced variants of differential privacy like (ϵ, δ)-differential privacy and Rényi differential privacy.

We describe our approach using the Haskell implementation of Solo, and demonstrate its use to verify differential privacy for practical algorithms in three case studies. We formalize a subset of Solo's sensitivity analysis and prove *metric preservation*, the soundness property for this analysis. Our implementation of Solo is available as open source [Abuah et al. 2022].[1]

Contributions. In summary, we make the following contributions:

- We introduce Solo, a novel type system for the static verification of differential privacy without linear types (§4).
- We present a reference implementation of Solo as a Haskell library, which retains support for type inference and does not require additional language extensions (§5, §6).
- We demonstrate the applicability of Solo by describing its standard library and three case studies (§7, §8).
- We formalize a subset of Solo's type system and prove its soundness (§9).

2 BACKGROUND

This section provides a summary of the fundamentals of differential privacy. Differential privacy [Dwork et al. 2006] affords a notion of *plausible deniability* at the individual level to participants in aggregate data analysis queries. In principle, a differentially private algorithm $\mathcal{K}$ over several individuals must include enough random noise to make the participation (removal/addition) of any one individual statistically unrecognizable. While this guarantee is typically in terms of a *symmetric difference* of one individual, formally a distance metric between datasets d is specified.

Definition 2.1 (Differential privacy). *For a distance metric $d_A \in A \times A \to \mathbb{R}$, a randomized* mechanism $\mathcal{K} \in A \to B$ *is (ϵ, δ)-differentially private if $\forall x, x' \in A$ s.t. $d_A(x, x') \leq 1$, considering any set S of possible outcomes, we have that:* $\Pr[\mathcal{K}(x) \in S] \leq e^{\epsilon} \Pr[\mathcal{K}(x') \in S] + \delta$.

We say that two inputs x and x' are *neighbors* when $d_A(x, x') = 1$. To provide meaningful privacy protection, two neighboring inputs are normally considered to differ in the data of a single individual. Thus, the definition of differential privacy ensures that the probability distribution over $\mathcal{K}$'s outputs will be roughly the same, whether or not the data of a single individual is included in the input. The strength of the guarantee is parameterized by the *privacy parameters* ϵ and δ. The case when $\delta = 0$ is often called *pure* ϵ-differential privacy; the case when $\delta > 0$ is often called *approximate* or (ϵ, δ)-differential privacy.

Sensitivity. The core mechanisms for differential privacy (described below) rely on the notion of *sensitivity* [Dwork et al. 2006] to determine how much noise is needed to achieve differential privacy.

[1] Most recent version available at: https://github.com/uvm-plaid/solo

Intuitively, function sensitivity describes the rate of change of a function's output relative to its inputs, and is a scalar value that bounds this rate, in terms of some notion of distance. Formally:

Definition 2.2 (Global Sensitivity). *Given distance metrics d_A and d_B, a function $f \in A \to B$ is said to be s-*sensitive *if $\forall s' \in \mathbb{R},\ (x, y) \in A.\ d_A(x, y) \leq s' \implies d_B(f(x), f(y)) \leq s' \cdot s$.*

For example, the function $\lambda x : \mathbb{R}.\ x + x$ is 2-sensitive, because its output is twice its input. Determining tight bounds on sensitivity is often the key challenge in ensuring differential privacy for complex algorithms.

Core Mechanisms. The core mechanisms that are often utilized to achieve differential privacy are the *Laplace mechanism* [Dwork et al. 2014a] and the *Gaussian mechanism* [Dwork et al. 2014a]. Both mechanisms are defined for scalar values as well as vectors; the Laplace mechanism requires the use of the L_1 distance metric and satisfies ϵ-differential privacy, while the Gaussian mechanism requires the use of the L_2 distance metric (which is often much smaller than L_1 distance) and satisfies (ϵ, δ)-differential privacy (with $\delta > 0$).

Definition 2.3 (Laplace Mechanism). *Given a function $f : A \to \mathbb{R}^d$ which is s-sensitive under the L_1 distance metric $d_{\mathbb{R}}(x, x') = \| x - x' \|_1$ on the function's output, the Laplace mechanism releases $f(x) + Y_1, \ldots, Y_d$, where each of the values $Y_1, \ldots, Y_d$ is drawn from the Laplace distribution centered at 0 with scale $\frac{s}{\epsilon}$; it satisfies ϵ-differential privacy.*

Definition 2.4 (Gaussian Mechanism). *Given a function $f : A \to \mathbb{R}^d$ which is s-sensitive under the L_2 distance metric $d_{\mathbb{R}}(x, x') = \| x - x' \|_2$ on the function's output, the Gaussian mechanism releases $f(x) + Y_1, \ldots, Y_d$, where each of the values $Y_1, \ldots, Y_d$ is drawn from the Gaussian distribution centered at 0 with variance $\sigma^2 = \frac{2s^2 \ln(1.25/\delta)}{\epsilon^2}$; it satisfies (ϵ, δ)-differential privacy for $\delta > 0$.*

Composition. Multiple invocations of a privacy mechanism on the same data degrade in an additive or compositional manner. For example, the law of *sequential composition* states that:

Theorem 2.1 (Sequential Composition). *If two mechanisms $\mathcal{K}_1$ and $\mathcal{K}_2$ with privacy costs of (ϵ_1, δ_1) and (ϵ_2, δ_2) respectively are executed on the same data, the total privacy cost of running both mechanisms is $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$.*

For iterative algorithms, *advanced composition* [Dwork et al. 2014a] can yield tighter bounds on total privacy cost. Advanced variants of differential privacy, like Rényi differential privacy [Mironov 2017] and zero-concentrated differential privacy [Bun and Steinke 2016], provide even tighter bounds on composition. We discuss composition in detail in Section 6.

Type Systems for Differential Privacy. The first static approach for verifying differential privacy in the context of higher-order programming constructs was Fuzz [Reed and Pierce 2010]. Fuzz uses linear types to verify both sensitivity and privacy properties of programs, even in the context of higher-order functions. Conceptual descendents of Fuzz include DFuzz [Gaboardi et al. 2013], Adaptive Fuzz [Winograd-Cort et al. 2017], Fuzzi [Zhang et al. 2019], Duet [Near et al. 2019], and the system due to Azevedo de Amorim et al. [de Amorim et al. 2019]. Approaches based on linear types combine a high degree of automation with support for higher-order programming, but require the host language to support linear types, so none has yet been implemented in a mainstream programming language.

Our work is closest to DPella [Lobo-Vesga et al. 2020], a Haskell library that uses the Haskell type system for sensitivity analysis. DPella implements a custom dynamic analysis of programs to compute privacy and accuracy information. Solo goes beyond DPella by supporting calculation of privacy costs using Haskell's type system, in addition to sensitivity information, and we have formalized its soundness. See Section 10 for a complete discussion of related work.

3 OVERVIEW OF SOLO

Solo is a static analysis for differential privacy, which can be implemented as a library in Haskell. Its analysis is completely static, and it does not impose any runtime overhead. Solo requires special type annotations, but in many cases these types can be inferred, and typechecking is aided by the flexibilty of parametric polymorphism in Haskell. Solo retains many of the strengths of linear typing approaches to differential privacy, while taking a light-weight approach capable of being embedded in mainstream functional languages. Specifically, Solo:

(1) is capable of sensitivity analysis for general-purpose programs in the context of higher order programming.
(2) implements a privacy verification approach with separate privacy cost analysis for multiple program inputs using ideas from Duet.
(3) leverages type-level dependency on values via Haskell singleton types, allowing verification of private programs with types that reference symbolic parameters
(4) features verification of several recent variants of differential privacy including (ϵ, δ) and Rényi differential privacy.

However, Solo is not intended for the verification of low-level privacy mechanisms such as the core mechanisms described previously, the exponential mechanism [Dwork et al. 2014a], or the sparse vector technique [Dwork et al. 2014a].

A Departure From Linear Types. Linear types have previously been used to track the consumption of finite resources, such as memory, in computer programs. They have also seen popular use in differential privacy analysis to track program sensitivity and the privacy budget expenditure. Linear types are attractive for such applications because they provide a strategy rooted in type theory and linear logic for tracking resources throughout the semantics of a core lambda calculus. However, while linear types are a natural fit for differential privacy analysis, implementations of linear type systems are not commonly available in mainstream programming languages, and when available are usually not sophisticated enough to support differential privacy analysis. In order to facilitate an approach to static language-based privacy analysis in a mainstream programming language, we have chosen to depart from a linear types based strategy, instead favoring an approach similar to static taint analysis.

This design decision has one huge advantage: it **enables verifying differential privacy in Haskell**, which does not provide linear types. It also brings several drawbacks, outlined below and detailed later in the paper:

- *Functions*: Linear typing provides an explicit type for sensitive functions, indicating the resource expenditure incurred if the function is called with certain arguments. Without linear types baked into a programming language, it is usually impossible to annotate function types in the required manner. However, as we will see later on, it is possible to bypass this limitation using polymorphism (see Section 5.3).
- *Recursion*: In addition to resource tracking for function introduction, linear type systems also provide a strategy for tracking resource usage during function elimination while accounting for self-referential functions (recursion). One example of this is a verified implementation of the `map` function. However, without linear types we must rely on trusted primitives in order to perform looping on our private programs (see Section 5.4).
- *Decisions & Branching*: Programs with linear type systems use annotated sum types and modified typing rules for **`case`** branching in order to preserve soundness. While working from the outside, building an analysis system as a library on top of a mainstream language, we are unable to modify the typing of **`case`** statements, and instead impose constraints on branching. Specifically,

we disallow branching on sensitive information (which does not restrict the set of private programs we can write) and a case analysis which returns sensitive information (or a non-deterministic value due to invocation of a privacy mechanism) must have the same sensitivity (or privacy cost) in each `case` alternative (see Section 5.5).

The Challenge of Sensitivity Analysis without Linear Types. Linear type systems track resources by attaching resource usages to individual program variables in type derivations. Without linear types, program variables are not typically available in function types—so without linear types, *where do we attach sensitivities?* Previous *dynamic* sensitivity analyses [Abuah et al. 2021b; Ebadi and Sands 2015; McSherry 2009; Zhang et al. 2018] have attached sensitivities to *values*. This approach works extremely well in a dynamic analysis, where functions are effectively inlined, so higher-order programming is easy to support.

Our static setting is more complicated. We embed sensitivities in base types—the static equivalent of the dynamic strategy of attaching sensitivities to values. This approach stands in contrast to the linear-types strategy of embedding sensitivities in function types. A naive implementation of our approach effectively prevents higher-order programming, since it is impossible to give sufficiently general types to sensitive functions. In summary, we faced four major challenges in developing Solo (we list the section describing our solution in parens):

(1) Encoding sensitivity and privacy accounting using limited type-level arithmetic (§5, §6)
(2) Developing and encoding metrics and sensitivities for base types, including lists (§5)
(3) Developing a minimal set of built-in primitives for programming in Solo, and implementing a standard library of useful functions using those primitives (§7)
(4) Developing a minimal set of correct identities to enable programming in Solo (§7, §8)

Threat Model. The threat model for Solo is "honest but fallible"—that is, we assume the programmer *intends* to write a differentially private program, but may make mistakes. Our approach implements a sound analysis for sensitivity and privacy, but its embedding in a larger system (Haskell) may result in weak points that a malicious programmer could exploit to subvert Solo's guarantees (unsoundness in Haskell's type system, for example). The Solo library can be used with Safe Haskell [Terei et al. 2012] to address this issue; Solo exports only a set of safe primitives which are designed to enforce privacy preserving invariants that adhere to our metatheory. Used with Safe Haskell, we are not aware of any way for a malicious programmer to subvert Solo's guarantee. Our guarantees against malicious programmers are therefore similar to those provided by language-based information flow control libraries that use Safe Haskell (e.g. [Russo et al. 2008]).

Solo's protection against malicious programmers relies on hiding constructors for sensitive datatypes (including Solo's *privacy monad*); programmers can *only* use specific primitive building blocks exposed by the Solo library to construct new functions with sensitivity or privacy guarantees. For example, Solo's `laplace` function implements the Laplace mechanism; it is not possible to write a different function with the same type (without using `laplace` itself) because the constructor for the privacy monad is not exposed by library.

Soundness. We formalize our privacy analysis in terms of a metric preservation metatheory and prove its soundness in Section 9 via a step-indexed logical relation w.r.t. a step-indexed big-step semantics relation. A consequence of metric preservation is that well-typed pure functions are semantically *sensitive* functions, and that well-typed monadic functions are semantically *differentially private* functions. Our model includes two variants of pair and list type connectives—one sensitive and the other non-sensitive—as well as recursive functions.

4 AVOIDING LINEAR TYPES: FROM FUZZ TO SOLO

This section introduces the usage of Solo based on code examples written in our Haskell reference implementation, and compares Solo to related techniques based on linear types.

Sensitivity Analysis. Consider the function $\lambda x : \mathbb{R}.\ x + x$ from Section 2, which is 2-sensitive in its argument x (when the *absolute difference* metric $|x - y|$ is used to measure the distance between inputs). The Fuzz language gives this function the type $\mathbb{R} \multimap_2 \mathbb{R}$, which encodes its sensitivity directly via an annotation on the linear function connective $\multimap$. The linear type systems of Fuzz, DFuzz, Fuzzi, Duet, and Amorim et al. contain typing rules like the following:

$$\textsc{t-var}\ \frac{s \geq 1}{\Gamma, x :_s \tau \vdash x : \tau} \qquad \textsc{t-splus}\ \frac{\Gamma_1 \vdash e_1 : \mathbb{R} \qquad \Gamma_2 \vdash e_2 : \mathbb{R}}{\Gamma_1 + \Gamma_2 \vdash e_1 + e_2 : \mathbb{R}} \qquad \textsc{t-lam}\ \frac{\Gamma, x :_s \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\ e : \tau_1 \multimap_s \tau_2}$$

The t-var rule says that each *use* of a program variable incurs a "cost" of 1 to total sensitivity, and the t-lam rule translates the sensitivity analysis results on the function's body into a sensitivity annotation on the function type. Here, the context Γ maps program variables to types *and sensitivities*. In linear type systems for differential privacy, the context Γ acts as both a type environment and a *sensitivity environment*. Rules like t-splus add together the sensitivity environments of their subexpressions—an operation that sums each variable's sensitivities pointwise (so $\{x :_1 \mathbb{R}\} + \{x :_1 \mathbb{R}\} = \{x :_2 \mathbb{R}\}$). Using these rules, we can write down the following derivation for the function $\lambda x : \mathbb{R}.\ x + x$:

$$\frac{\dfrac{\{x :_1 \mathbb{R}\} \vdash x : \mathbb{R} \qquad \{x :_1 \mathbb{R}\} \vdash x : \mathbb{R}}{\{x :_2 \mathbb{R}\} \vdash x + x : \mathbb{R}}}{\{\} \vdash \lambda x : \mathbb{R}.\ x + x : \mathbb{R} \multimap_2 \mathbb{R}}$$

Sensitivity tracking in a linear type system is fundamentally linked to sensitivity environments mapping program variables to sensitivities, and is modeled as a co-effect (i.e. sensitivity environments are part of the context Γ).

Sensitivity in Solo. In Solo, we instead attach sensitivity environments to *base types*. Our sensitivity environments associate sensitivities with *data sources* (a set of global variables specified by the programmer, detailed in Section 5.1). For example, we can define a function that doubles its argument as follows:

```
dbl :: SDouble Diff senv -> SDouble Diff (senv +++ senv)
dbl x = x <+> x
```

We define *sensitive base types*, like `SDouble`, which augment base types with a distance metric (§5.1) and a sensitivity environment (denoted `senv`). `Diff` indicates the *absolute value metric*, which calculate the distance between two inputs as the absolute value of their difference. The sensitivity environment `senv` in the type of `dbl` plays the same role as the sensitivity annotations in the context Γ in the linear typing rules above. The `+++` operator adds two sensitivity environments elementwise; adding a sensitivity environment to itself doubles the sensitivity of each element in the environment. The `<+>` function is built into Solo, and its type mirrors the t-splus rule above:

```
(<+>) :: SDouble m s1 -> SDouble m s2 -> SDouble m (s1 +++ s2)
```

Functions in Solo have regular function types ($\tau_1 \longrightarrow \tau_2$), and sensitivity environments are attached only to base types. Without polymorphism, this difference leads directly to a significant loss of expressive power: without polymorphism, the type of `dbl` would need to specify exactly what data sources are defined for the program, and how sensitive the function's *input* is with respect to each one. Even *with* polymorphism, there are some functions (like `map`) for which linear-type-based approaches provide more general types. We detail the interaction between polymorphism and sensitivity environments in Section 5.3.

Privacy Analysis. Sensitivity tells us how much noise we need to add to a particular value to achieve the definition of differential privacy. To determine the total privacy cost of a complete program, we need to use the sequential composition property of differential privacy. Languages based on linear types include a language fragment for operations on differentially private values (often in the form of a *privacy monad*), with typing rules like the following:

$$\textsc{t-laplace}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \sqsubseteq \rceil\Gamma\lceil^{s}}{\rceil\Gamma\lceil^{\epsilon} \vdash \texttt{laplace}[s,\epsilon](e) : \bigcirc\tau} \qquad \textsc{t-bind}\ \frac{\Gamma_1 \vdash e_1 : \bigcirc\tau_1 \qquad \Gamma_2, x :_{\infty} \tau_1 \vdash e_2 : \bigcirc\tau_2}{\Gamma_1 + \Gamma_2 \vdash x \leftarrow e_1 \,;\, e_2 : \bigcirc\tau_2}$$

The notation $\bigcirc\tau$ denotes differentially private values. The t-laplace rule says that the `laplace` function (§2, Definition 2.3) satisfies ϵ-differential privacy, and returns a differentially private value. $\rceil\Gamma\lceil^{s}$ denotes the *truncation* of the context Γ to the sensitivity s (i.e. replacing every sensitivity in Γ with s). $\Gamma \sqsubseteq \rceil\Gamma\lceil^{s}$ encodes the requirement from Definition 2.3 that the argument to the Laplace mechanism must be at most s-sensitive, and $\rceil\Gamma\lceil^{\epsilon}$ replaces each sensitivity in the context with the privacy cost ϵ. The t-bind rule encodes sequential composition (§2, Theorem 2.1), adding up the privacy costs of both computations. For example, the rules above can show that the program $\texttt{laplace}[2,\epsilon](x + x)$ satisfies ϵ-differential privacy:

$$\dfrac{\dfrac{\{x :_1 \mathbb{R}\} \vdash x : \mathbb{R} \qquad \{x :_1 \mathbb{R}\} \vdash x : \mathbb{R}}{\{x :_2 \mathbb{R}\} \vdash x + x : \mathbb{R}}}{\{x :_{\epsilon} \mathbb{R}\} \vdash \texttt{laplace}[2,\epsilon](x + x) : \bigcirc\mathbb{R}}$$

In Fuzz's privacy monad, the context Γ associates *privacy costs* (rather than sensitivities) with program variables. In our example, the contexts in the first and second rows of the derivation contain sensitivities, while the context in the bottom row contains privacy costs. Linear function types can also encode privacy costs; the Laplace mechanism, for example, can be given the type $\mathbb{R} \multimap_{\epsilon} \bigcirc\mathbb{R}$. Conflating sensitivity and privacy this way works well for pure ϵ-differential privacy, but does not work for variants like (ϵ, δ)-differential privacy; recent linear type systems that support these variants (e.g. [de Amorim et al. 2019; Near et al. 2019]) are more complex as a result.

Privacy in Solo. In Solo, we attach privacy costs (in the form of *privacy environments*) to monadic values. We define a privacy monad in Haskell (denoted `PM`, detailed in Section 6) for which the `bind` operator adds privacy environments in the same way as the t-bind rule above. The Laplace mechanism has the following type:

```
laplace :: forall ϵ s. (TL.KnownNat (MaxSens s)) =>
  SDouble Diff s -> PM (TruncatePriv ϵ Zero s) Double
```

Here, `MaxSens` is a type-level operation corresponding to $\Gamma \sqsubseteq \rceil\Gamma\lceil^{s}$ in the t-laplace rule above (i.e. it ensures the maximum sensitivity of the mechanism's input is s), and `TruncatePriv` is a type-level operation corresponding to $\rceil\Gamma\lceil^{\epsilon}$ (i.e. it converts the sensitivity environment `senv` to a privacy environment). `TL.KnownNat` is a type-level constraint that allows *runtime* access to the maximum sensitivity of the type-level sensitivity environment `s`. This value is required for the `laplace` function to correctly sample noise at runtime. The type `Zero` is a type-level real-number representation of 0. The following function takes a `SDouble` as input, doubles it, and applies the Laplace mechanism:

```
simplePrivacyFunction :: TL.KnownNat (MaxSens (s +++ s)) =>
  SDouble Diff s -> PM (TruncatePriv (RNat 2) Zero (s +++ s)) Double
simplePrivacyFunction x = laplace @(RNat 2) (dbl x)
```

The type `PM (TruncatePriv (RNat 2) Zero (s +++ s)) Double` indicates that the function satisfies $(\epsilon, \delta$-differential privacy for $\epsilon = 2$ and $\delta = 0$, for all of the data sources mentioned in the sensitivity environment `s`. `RNat` and `Zero` are used for type-level real numbers, used to describe fractional privacy costs. We use Haskell's type-level application operator `@` in the call to `laplace` to specify

```
import qualified GHC.TypeLits as TL

-- Sources & Sensitivity Environments (§5.1)
type Source = TL.Symbol
data Sensitivity = InfSens | NatSens TL.Nat
type SEnv = [(Source, Sensitivity)]

-- Distance Metrics (§5.1)
data NMetric = Diff | Disc
data CMetric = L1 | L2 | LInf

-- Doubles
SDouble :: NMetric -> SEnv -> *
```

```
-- Pairs (§5.2)
SPair    :: CMetric -> (SEnv -> *)
            -> (SEnv -> *) -> SEnv -> *
L1Pair   = SPair L1      -- ⊗-pairs in Fuzz
L2Pair   = SPair L2      -- Not in Fuzz
LInfPair = SPair LInf    -- &-pairs in Fuzz

-- Lists (§5.2)
SList    :: CMetric -> (SEnv -> *)
            -> SEnv -> *
L1List   = SList L1      -- τ list in Fuzz
L2List   = SList L2      -- Not in Fuzz
LInfList = SList LInf    -- τ alist in Fuzz
```

Fig. 1. Sensitivity Types in Solo.

the value for ϵ. As in the previous example, Haskell is able to infer the type if the annotation is left off. When the function is applied to a sensitive value with a concrete sensitivity environment, the maximum sensitivity of the argument to the Laplace mechanism is automatically calculated by `MaxSens`, and does not need to be specified by the programmer.

As with sensitivity analysis, we rely heavily on polymorphism to produce general types for functions that guarantee differential privacy (e.g. the `laplace` function). Absent polymorphism, the type for the Laplace mechanism would need to specify an exact set of data sources and a concrete privacy cost associated with each one.

5 SENSITIVITY ANALYSIS

Prior type-based analyses for sensitivity analysis [Gaboardi et al. 2013; Near et al. 2019; Reed and Pierce 2010; Winograd-Cort et al. 2017] focus on *function sensitivity* with respect to *program variables*. Solo's type system, in contrast, associates sensitivity with *base types* (not functions), and these sensitivities are determined with respect to *data sources* (not program variables). This difference represents a significant departure from previous systems, and is the key design feature that enables embedding Solo's type system in a language (like Haskell) without linear types. Figure 1 presents the types for the sensitivity analysis in the Solo system. The rest of this section describes types in Solo and how they can be used to describe the sensitivity of a program. We describe the privacy analysis in Section 6, and we formalize both analyses in Section 9.

5.1 Types, Metrics, and Environments

This section describes Figure 1 in detail. We begin with sources (written o), environments (Σ), metrics (m and w), types (τ), and sensitive types (σ).

Sources & Environments. Our approach makes use of the idea that a static privacy analysis of a program can be centered around a global set of sensitive *data sources* which the analyst wants to preserve privacy for. Data sources are represented by type-level symbols, each of which represents a single sensitive program input (e.g. raw numeric data, a file or an IO stream). In the most common case, when data is read from a file, the source is identified by the data's filename. Solo's data sources are inspired by ideas from static taint analysis—we "taint" the program's data sources with sensitivity annotations that are tracked and modified throughout type-checking. Solo tracks sensitivity *relative* to data sources (i.e. Solo assumes that data sources have an "absolute sensitivity" of 1). In Solo, like in Fuzz, *sensitivities* can be either a number or ∞. In Solo, numeric sensitivities are represented using type-level natural numbers. A *sensitivity environment* `SEnv` is an association list of data sources and their sensitivities, and corresponds to the same concept in Fuzz.

Distance Metrics & Metric-Carrying Types. Interpreting sensitivity requires describing how to measure distances between values (as described in Definition 2.2); different metrics for this measurement produce different privacy properties. Solo provides support for several distance metrics including those commonly used in differentially private algorithms. The *base metrics* listed in Figure 1 (denoted `NMetric`) are distance metrics for base types. The *sensitive base types* are metric-carrying base types (i.e. every sensitive type must have a distance metric). For example, the type of a sensitive `Double` is `SDouble m`, where `m` is a metric. The base metrics are `Diff`, the *absolute difference metric* ($d(x,y) = |x-y|$), and `Disc`, the *discrete metric* ($d(x,y) = 0$ *if* $x = y; 1$ *otherwise*). Thus the types `SDouble Diff` and `SDouble Disc` mean very different things when interpreting sensitivity. The distance between two values v_1, v_2 : `SDouble Diff` is $|v_1 - v_2|$, but the distance between two values v_3, v_4 : `SDouble Disc` is at most 1 (when $v_3 \neq v_4$).

Both of these metrics are useful in writing differentially private programs; basic mechanisms for differential privacy (like the Laplace mechanism) typically require their inputs to use the `Diff` metric, while the distance between program inputs is often described using the `Disc` metric. For example, we might consider a "database" of real numbers, each contributed by one individual; two neighboring databases in this setting will differ in exactly one of those numbers, but the change to the number itself may be unbounded. In this case, each number in the database would have the type `SDouble Disc`. Fuzz fixes the distance metric for numbers to be the absolute difference metric; Duet provides two separate types for real numbers, each with its own distance metric.

Types. A sensitive type in Solo carries both a metric and a sensitivity environment (e.g. `SDouble` has kind `NMetric -> SEnv -> *`). Thus, sensitivities are associated with *values*, rather than with *program variables* (as in Fuzz). For example, the type `SDouble Diff '[ '("input", 1) ]` is the type of a double value with a concrete sensitivity environment, stating the value is is 1-sensitive with respect to the data source *input* under the absolute difference metric. Adding such a value to itself results in the type `SDouble Diff '[ '("sensitive_input", 2) ]`—encoding the fact that the sensitivity has doubled. In Fuzz, the same information is encoded by the sensitivities recorded in the context; but with respect to program variables rather than data sources. Note that it is not possible to attach a sensitivity environment to a function type—*only* the metric-carrying sensitive types may have associated sensitivity environments. Solo does not provide a "sensitive function" type connective (like Fuzz's $\multimap$); in Solo, function sensitivity must be stated in terms of the sensitivity of the function's arguments with respect to the program's data sources (more in Section 5.3).

Operations on Sensitivity Environments. Section 4 describes several type-level functions on sensitivity environments in Solo, including `+++`, `MaxSens`, `TruncateSens`, and `TruncatePriv`. We implement these functions in Solo as Haskell type families. For example, the definition of `MaxSens` appears below.

```
type family MaxSens (s :: SEnv) :: TL.Nat where
  MaxSens '[] = 0
  MaxSens ('(_, NatSens n)':s) = MaxNat n (MaxSens s)
```

The other operations are similarly defined as simple recursive functions at the type level, which mimic the mathematical definitions used earlier and in our formalism (§9). The definition of `+++` is slightly more complicated, because `+++` must find matching sources in its two input environments and add their sensitivities. To make this possible, we ensure that sensitivity environments are ordered by their keys (the symbols representing data sources), and define operations like `+++` to maintain that ordering. These functions appear in the supplemental material [Abuah et al. 2021a] .

5.2 Pairs and Lists

The Fuzz system contains two connectives for pairs, $\otimes$ and $\&$, which differ in their metrics. The distance between two $\otimes$ pairs is the sum of the distances between their elements, while the distance between two $\&$ pairs is the maximum of distances between their elements. Solo provides a single pair type, `SPair`, that can express both types by specifying a *compound metric* `CMetric`.

Compound Metrics. In Solo, metrics for compound types are derived from standard vector-space distance metrics. For example, a sensitive pair has the type `SPair w` where `w` is one of the compound metrics in Figure 1 (e.g. `L1`, the L_1 (or *Manhattan*) distance; `L2`, the L_2 (or *Euclidian*) distance; or `LInf`, the L_∞ distance). Thus we can represent Fuzz's $\otimes$ pairs in Solo using the `SPair L1` type constructor, and Fuzz's $\&$ pairs using `SPair LInf`. We can construct pairs from sensitive values using the following two functions:

```
makeL1Pair :: a m s₁ -> b m s₂ -> SPair L1 a b (s₁ +++ s₂)        -- Fuzz's ⊗-pair
makeLInfPair :: a m s₁ -> b m s₂ -> SPair LInf a b (Join s₁ s₂)    -- Fuzz's &-pair
```

Here, the `+++` operator for sensitivity environments performs elementwise addition on sensitivities, and the `Join` operator performs elementwise maximum.

Lists. Fuzz defines the list type τ `list`, and gives types to standard operators over lists reflecting their sensitivities. In Solo, we define the `SList` type to represent sensitive lists. Sensitive lists in Solo carry a metric, in the same way as sensitive pairs, and can only contain metric-carrying types. The type of a sensitive list of doubles with the L_1 distance metric, for example, is `SList L1 SDouble`; this type corresponds to Fuzz's $\otimes$-lists. The type `SList LInf SDouble` corresponds to Fuzz's $\&$-lists. Fuzz does not provide the equivalent of `SList L2 SDouble`, which uses the L_2 distance metric.

The distance metrics available in Solo are useful for writing practical differentially private programs. For example, we might want to sum up a list of sensitive numbers drawn from a database. The typical definition of neighboring databases tells us that the distance between two such lists is equal to the number of elements which differ—and those elements may differ by any amount. As a result, their sums may also differ by any amount, and the sensitivity of the computation is unbounded. To address this problem, differentially private programs often *clip* (or "top-code") the input data, which enforces an upper bound on input values and results in bounded sensitivity. We use this technique in one of our case studies (Section 8).

5.3 Function Sensitivity & Higher-Order Functions

In Fuzz, an s-sensitive function is given the type $\tau_1 \multimap_s \tau_2$. Solo does not have sensitive function types, but we have already seen examples of the approach used in Solo to bound function sensitivity: we write function types that are polymorphic over sensitivity environments. In general, we can recover the notion of an s-sensitive function in Solo by writing a Haskell function type that scales the sensitivity environment of its input by a scalar s:

```
s_sensitive :: SDouble m senv -> SDouble m (ScaleSens senv s)   -- An s-sensitive function
```

Here, `ScaleSens` is implemented as a type family that scales the sensitivity environment `senv` by `s`: for each mapping $o \mapsto s_1$ in `senv`, the scaled sensitivity environment contains the mapping $o \mapsto s \cdot s_1$. The complete implementations of all type-level functions for sensitivity environments appear in the supplemental material [Abuah et al. 2021a] . The common case of a 1-sensitive (or linear) function can be represented by keeping the input's sensitivity environment unchanged (as in `clip` and `sum` in the previous section):

```
one_sensitive :: SDouble m senv -> SDouble m senv                -- A 1-sensitive function
```

Sensitive Higher-Order Operations An important goal in the design of Solo is support for sensitivity analysis for higher-order, general-purpose programs. For example, prior systems such as Fuzz and Duet encode the type for the higher-order map function as follows:

$$\text{map} : (\tau_1 \multimap_s \tau_2) \multimap_\infty \text{list}\ \tau_1 \multimap_s \text{list}\ \tau_2$$

This map function describes a computation that accepts as inputs: an s-sensitive unary function from values of type τ_1 to values of type τ_2 (map is allowed to apply this function an unlimited number of times), and a list of values of type τ_1. map returns a list of values of type τ_2 which is s-sensitive in the former list. We can give an equivalent type to smap in Solo as follows, by explicitly scaling the appropriate sensitivity environments using type-level arithmetic:

```
smap :: forall fn_sens a b s2 m. (forall s1. a s1 -> b (ScaleSens s1 fn_sens))
  -> SList m a s2 -> SList m b (ScaleSens s2 (MaxNat fn_sens 1))
```

Polymorphism for Sensitive Function Types. Special care is needed for functions that close over sensitive values, especially in the context of higher-order functions like smap. Consider the following example:

```
dangerousMap :: SDouble m1 s1 -> SList m2 (SDouble m1) s2 -> _
dangerousMap x ls = let f y = x in smap f ls
```

Note that f is *not* a function that is s-sensitive with respect to its input—instead, it is s_1-sensitive with respect to the closed-over value of x. This use of map is dangerous, because it may apply f many times, creating duplicate copies of x without accounting for the sensitivity effect of this operation. Fuzz assigns an infinite sensitivity for x in this program. Solo rejects this program as not well-typed. The type of f is `SDouble m1 s3 -> SDouble m1 s1`, but smap requires it to have the type `(forall s1. a s1 -> b (ScaleSens s1 fn_sens))`—and these two types do not unify. Specifically, the scope of the sensitivity environment s' is limited to f's type—but in the situation above, the environment s1 comes from *outside* of that scope.

The use of parametric polymorphism to limit the ability of higher-order functions to close over sensitive values is key to our ability to support this kind of programming. Without it, we would not be able to give a type for map that ensures soundness of the sensitivity analysis. The use of parametric polymorphism to aid in information flow analysis has been previously noted [Bowman and Ahmed 2015], and is also key to the treatment of sensitivity in DPella [Lobo-Vesga et al. 2020].

5.4 Recursion

In Fuzz, it is possible not only to write the type of *map*, but to *infer* it from the definition. The Fuzz type system contains general recursive datatypes, and its typing rules admit recursive programs over those datatypes (like *map*). Solo's sensitive list types are less powerful than Fuzz's. In Solo, it is possible to give types to recursive functions over lists (like map, as seen in Section 5.3). However, it is not possible to typecheck the *implementations* of these functions using Solo's types, since the structure of a sensitive list is opaque to programs written using the Solo library. We provide *fold* as a build-in primitive, and use it to define other recursive functions. See Section 7 for more.

5.5 Conditionals

Sensitivity analysis for conditionals requires care, whether or not linear types are used. The primary challenge is that branching on a sensitive value typically implies *infinite* sensitivity with respect to its variables, since the resulting control flow reveals information about the condition. Systems

like Fuzz handle this problem using a CASE rule that scales the sensitivity environment used to typecheck condition by the number of times the result is used in the two branches. In practice, this approach often disallows branching on sensitive values, since the distance metric for boolean values says that true and false are infinitely far apart.

In Solo, as in many systems for static information flow control [Myers 1999], we disallow branching on sensitive values altogether. This restriction is implemented implicitly by the opacity of sensitive types (e.g. it is not possible to compare two `SDouble` values and obtain a boolean, so branching on `SDouble` values is impossible). This restriction does not significantly limit the set of differentially private programs that we can write with Solo; except for special mechanisms like the Sparse Vector Technique [Dwork et al. 2014a], differentially private algorithms generally do not branch on sensitive values anyway, because doing so would violate privacy.

Solo's sensitive datatypes are opaque, and their primitive constructors are hidden when exporting the types to other modules. Solo exports only safe operations with known sensitivity properties for sensitive datatypes. This design makes it impossible to branch directly on sensitive datatypes; for example, the following attempt fails to typecheck, because equality is not defined on sensitive doubles:

```
tryEquals :: SDouble Diff s1 -> Bool
tryEquals x = x == sConstD 0
```

```
- No instance for (Eq (SDouble 'Diff s1))
  arising from a use of '=='
- In the expression: x == sConstD 0
  In an equation for 'tryEquals': tryEquals x = x == sConstD 0
```

It *is* possible (and desirable) to allow branching on differentially private values (i.e. noisy values). The basic mechanisms in Solo (e.g. `laplace`) return regular Haskell values (e.g. `Double` instead of `SDouble`), and branching on these values can be done in the usual way.

6 PRIVACY ANALYSIS

The goal of static privacy analysis is to check that (1) the program adds the correct amount of noise for the sensitivity of underlying computations (i.e. that core mechanisms are used correctly), and (2) the program composes privacy-preserving computations correctly (i.e. the total privacy cost of the program is correct, according to differential privacy's composition properties). A well-typed program should satisfy both conditions. As described earlier, sensitivity analysis often supports privacy analysis, especially in systems based on linear types.

Previous work has taken several approaches to static privacy analysis; we provide a summary in the next section. Solo provides a *privacy monad* that encodes privacy as an effect. As in our sensitivity analysis, the primary difference between Solo and previous work is that our privacy monad tracks privacy cost with respect to data sources, rather than program variables. This distinction allows the implementation of Solo's privacy monad in Haskell, and additionally enables our approach to describe variants of differential privacy without linear group privacy (e.g. (ϵ, δ)-differential privacy).

6.1 Existing Approaches for Privacy Analysis

The Fuzz language pioneered static verification of ϵ-differential privacy, using a linear type system to track sensitivity of data transformations. In this approach, the linear function space can be interpreted as a space of ϵ-differentially private functions by lifting into the probability monad. However, more advanced variants of differential privacy such as (ϵ, δ) differential privacy do not satisfy the restrictions placed on the interpretation of the linear function space in this approach, and

Fuzz cannot be easily extended to support these variants. Azevedo de Amorim et al. [de Amorim et al. 2019] provide an extensive discussion of this challenge.

More recently, Lobo-Vesga et al. in DPella [Lobo-Vesga et al. 2020] present an approach in Haskell which tracks sensitivity via data types which are indexed with their *accumulated stability* i.e. sensitivity. Typically in privacy analysis we consider sensitivity to be a property of functions, however as they show, we can also represent sensitivity via the arguments to these functions. Their approach represents private computations via a monad value and monadic operations, similar to the approach in Fuzz. However, in the absence of true linear types, their approach relies on dynamic taint analysis and runtime symbolic execution.

The technique of separating sensitivity composition from privacy composition has been seen before, subsequent to Fuzz, in order to facilitate (ϵ, δ)-differential privacy. Azevedo de Amorim et al. [de Amorim et al. 2019] introduce a *path construction* technique which performs a *parameterized comonadic lifting* of a metric space layer à la Fuzz to a separate relational space layer for (ϵ, δ) differential privacy. The Duet system [Near et al. 2019] uses a dual type system, with dedicated systems for sensitive composition and privacy composition. In principle, this follows a combined effect/co-effect system approach [Petricek 2017], where one type system tracks the co-effect (in this case sensitivity) and another tracks the effect which is randomness due to privacy.

Our approach embodies the spirit of Duet and simulates coeffectful program behavior by embedding the co-effect (i.e. the entire sensitivity environment) as an index in comonadic base data types. We then track privacy composition via a special monadic type as an effect. As in Duet, the core privacy mechanisms such as Laplace and Gauss police the boundary between the two. Due to the nature of our co-effect oriented approach in which we track the full sensitivity context, our solution can be embedded in Haskell completely statically, without the need for runtime dynamic symbolic execution. We are also able to verify advanced privacy variants such as (ϵ, δ) and state-of-the-art composition theorems such as advanced composition and the moments accountant via a family of higher-order primitives.

Monads & Effect Systems. Effect systems are known for providing more detailed static type information than possible with monadic typing. They are the topic of a variety of research on enhancing monadic types with program effect information, in order to provide stricter static guarantees. Orchard et al [Orchard et al. 2014], following up on initial work by Wadler and Thiemann [Wadler and Thiemann 2003], provide a denotational semantics which unify effect systems with a monadic-style semantics as a *parametric effect monad*, establishing an isomorphism between indices of the denotations and the effect annotations of traditional effect systems. They present a formulation of parametric effect monads which generalize monads to include annotation of an effect with a strict monoidal structure. Below typing rules of the general parametric effect monad are shown:

$$\textsc{Bind}\ \frac{f : a \to M\ r\ b \qquad g : b \to M\ s\ c}{\lambda x.\ f\ x >\!\!>\!\!= g : a \to M\ (r \otimes s)\ c} \qquad \textsc{Return}\ \frac{}{return : a \to M\ \varnothing\ a}$$

These typing rules describe a formulation of parametric effect monads M which accept an effect index as their first argument. This effect index of some arbitrary type E is a monoid $(E, \otimes, \varnothing)$.

6.2 Solo's Privacy Monad

Solo defines *privacy environments* in the same way as sensitivity environments; instead of tracking a sensitivity with respect to each of the program's data sources, however, a privacy environment tracks a *privacy cost* associated with each data source. Privacy environments for pure ϵ-differential privacy are defined as follows:

```
-- Privacy Environments
data TLReal = Lit Rat | TLReal :+: TLReal | TLReal :*: TLReal | Ln TLReal | Root TLReal | ...
type Zero = Lit (Rat_REDUCED 0 1)
type RNat n = Lit (Rat_REDUCED n 1)

type EDEnv = [(TL.Symbol, TLReal, TLReal)]
```

Type-level natural numbers were sufficient to represent sensitivities. Privacy costs, however, are often real numbers less than 1, and privacy cost expressions sometimes contain logarithms and square roots (as in advanced composition, described later). Haskell avoids supporting doubles at the type level, because native equality for doubles (e.g., NaN) does not interact well with the notion of equality required for typing. We therefore implement `TLReal` by building type-level expressions that represent real-valued computations, and interpret those expressions using Haskell's standard double type at the value level. We define `Zero` and `RNat` as shorthand for constants, and we define the privacy environment `EDEnv` (for ϵ, δ-env) as a mapping from sources to pairs of real expressions representing the ϵ and δ part of the privacy cost.

The *sequential composition* theorem for differential privacy (Theorem 2.1) says that when sequencing (ϵ, δ)-differentially private computations, we can add up their privacy costs. This theorem provides the basis for the definition of a privacy monad. We observe that our privacy environments have a monoidal structure (`EDEnv`, `(++++)`, `'[]`), where `++++` is a type family implementing the sequential composition theorem. We derive a privacy monad which is indexed by our privacy environments, in the same style as a notion of effectful monads or *parametric effect monads* given separately by Orchard [Orchard and Petricek 2014; Orchard et al. 2014] and Katsumata [Katsumata 2014]. Computations of type `PM` are constructed via these core functions:

```
-- Privacy Monad for (ϵ, δ)-differential privacy
newtype PM (p :: EDEnv) a = PM_UNSAFE { unPM :: IO a }

return :: a -> PM '[] a
return x = PM_UNSAFE $ P.return x

(>>=) :: PM p1 a -> (a -> PM p2 b) -> PM (p1 ++++ p2) b
xM >>= f = PM_UNSAFE $ unPM xM P.>>= (unPM . f)
```

The `return` operation accepts some value and embeds it in the `PM` without causing any side-effects. The `(>>=)` (or `bind`) operation allows us to sequence private computations using differential privacy's sequential composition property, encoded here as the type family `++++`. The implementation of `++++` performs elementwise summation of two privacy environments. In the computation `f>>=g` we execute the private computation `f` for some polymorphic privacy cost $\mathtt{p}_1$, pass its result to the private computation `g`, and output the result of `g` at a total privacy cost of the degradation of the $\mathtt{p}_1$ and $\mathtt{p}_2$ privacy environments combined according to sequential composition. `PM` is an indexed monad; we leverage Haskell's `RebindableSyntax` language extension to enable `do`-notation in our examples.

Note that `return` in Solo's privacy monad is very different from the same operator in Fuzz. The typing rule for `return` in Fuzz scales the sensitivities in the context by ∞—reflecting the idea that `return`'s argument is revealed with no added noise, incurring infinite privacy cost. Fuzz, $\mathtt{return}\ x \gg\!= \mathtt{laplace} \neq \mathtt{laplace}\ x$. In Solo, the `return` operator attaches an empty privacy environment to the returned value. If a sensitive value is given as the argument to `return`, then *it remains sensitive*, rather than being revealed (as in Fuzz)—so there is no need to assign the value an infinite privacy cost. This approach is not feasible in Fuzz because privacy costs are associated

with program variables rather than with values. We can recover Fuzz's `return` behavior with a function that reveals a value without noise, and scales its privacy cost by infinity.

Privacy Variants & Converting Between Variants. Our approach for privacy analysis can support any variant of differential privacy with sequential composition and post-processing theorems. Our current implementation includes privacy monads for both (ϵ, δ)-differential privacy and Rényi differential privacy (RDP) [Mironov 2017]. For each privacy variant, the corresponding privacy monad is indexed with a privacy environment that tracks the appropriate privacy parameters, and the bind operation enforces the appropriate form of sequential composition. Solo implements conversions between variants, to enable variant-mixing in programs—for example, to convert from Rényi differential privacy to (ϵ, δ)-differential privacy (as is common in machine learning algorithms):

```
conv_RDP :: RDP_PM p a -> PM (ConvRDPtoED p) a
```

6.3 Core Privacy Mechanisms

This section shows how we encode three core mechanisms for differential privacy in Solo: the Laplace mechanism, the Gaussian mechanism, and the exponential mechanism.

The Laplace mechanism. We can now define the Laplace mechanism (described in Section 2), which satisfies $(\epsilon, 0)$-differential privacy:

```
laplace :: forall eps s. (TL.KnownNat (MaxSens s)) =>
           SDouble Diff s -> PM (TruncatePriv eps Zero s) Double
laplaceL :: forall eps s. (TL.KnownNat (MaxSens s)) =>
            L1List (SDouble Diff) s -> PM (TruncatePriv eps Zero s) [Double]
```

The first argument to `laplace` is the privacy parameter ϵ (as a type-level parameter). The second argument is the value we would like to add noise to; it must be a sensitive number with the `Diff` metric. The function's result is a regular Haskell `Double`, in the privacy monad. The `TruncatePriv` type family transforms a sensitivity environment into a privacy environment by replacing each sensitivity with the privacy parameter ϵ. The function's implementation follows the definition of the Laplace mechanism; it determines the scale of the noise to add using the maximum sensitivity in the sensitivity environment `s` and the privacy parameter ϵ. The `laplaceL` function implements the *vector-valued Laplace mechanism*, which adds noise to each element of a vector based on the vector's L_1 sensitivity. Its argument is required to be a `L1List` of sensitive doubles with the `Diff` metric, and its output is a list of Haskell doubles in the privacy monad.

The Gaussian Mechanism. The Gaussian mechanism (described in Section 2) adds Gaussian noise instead of Laplace noise, and ensures (ϵ, δ)-differential privacy (with $\delta > 0$). The primary advantage of the Gaussian mechanism is in the vector setting: the Gaussian mechanism uses L_2 sensitivity, which is typically much lower than the L_1 sensitivity used by the Laplace mechanism. This requirement is reflected in the type of the Gaussian mechanism for lists in Solo:

```
gaussL :: forall ϵ δ n s. (TL.KnownNat (MaxSens s)) =>
  L2List (SDouble Diff) s -> PM (TruncatePriv ϵ δ s) [Double]
```

The Exponential Mechanism. We also implement the exponential mechanism [McSherry and Talwar 2007], which selects an element from a public set that approximately maximizes a programmer-defined *score function*. It satisfies $(\epsilon, 0)$-differential privacy.

```
expMech :: forall eps s1 t1 t2.
  (forall s. t1 -> t2 s -> SDouble Diff s) -> [t1] -> t2 s1 -> PM (TruncatePriv eps Zero s1) t1
```

We represent the public set as a list with elements of type `t1`; the score function returns a sensitive double, and must be 1-sensitive in its second argument. The output type indicates the privacy cost.

6.4 Looping Combinators

Like Fuzz, Solo does not support recursive functions that satisfy differential privacy, because computing an upper bound on privacy cost requires knowing how many times the function's body will execute. DFuzz uses dependent types to allow functions to be written recursively, while Duet defines looping combinators to allow for iterative algorithms. We adopt the latter approach, and define looping combinators for Solo.

Sequential composition. For an (ϵ, δ)-differentially private mechanism that runs in a loop for k iterations, the sequential composition theorem (Theorem 2.1) implies that the loop satisfies $(k\epsilon, k\delta)$-differential privacy. We encode these loops using the `seqloop` combinator:

```
seqloop :: (TL.KnownNat k) => (Int -> a -> PM p a) -> a -> PM (ScalePriv p k) a
```

Here, the loop body is written as a function whose inputs are the iteration number (as a public `Int`) and a value of type `a` representing the loop's current accumulator value. The loop body should have a privacy cost described by the type parameter `p`. The output type scales `p` (both the ϵ and δ parts) via `ScalePriv` by the number of iterations k. See the supplemental material [Abuah et al. 2021a] for the full definition of `ScalePriv` and other type-level functions for privacy environments.

Advanced composition. The *advanced composition theorem* for differential privacy [Dwork et al. 2014a] can provide tighter bounds on the privacy cost of iterative algorithms. It says that running an (ϵ, δ)-differentially private mechanism k times satisfies $(2\epsilon\sqrt{2k\ln(1/\delta')}, k\delta + \delta')$-differential privacy, for any $\delta' > 0$. The corresponding looping combinator is written as follows:

```
advloop :: forall k delta_prime p a.
  (TL.KnownNat k) => (Int -> a -> PM p a) -> a -> PM (AdvComp k delta_prime p) a
```

The `advloop` combinator has the same interface as `seqloop`, but uses the type family `AdvComp` to statically compute total privacy cost using advanced composition. `AdvComp` builds a type-level expression containing square roots and logarithms, which matches the expression given by the advanced composition theorem above. See the supplemental material [Abuah et al. 2021a] for the full definition.

7 THE SOLO LIBRARY

We have implemented Solo as a Haskell library in about 700 lines of code; it will be made open-source upon publication and will be submitted as an artifact. This section describes the primitives available for sensitive datatypes in Solo, and how they are used to implement Solo's standard library. We also provide identities that are useful for obtaining the desired sensitivity bounds on functions, and describe how to use them.

Sensitivity Primitives. Solo's built-in primitives for sensitive doubles and lists—with axiomatized sensitivity properties reflected in their types—appear in Figure 2. Like Fuzz, Solo has primitives for basic operations on numbers and lists (e.g. `<+>` and `scons`). Also like Fuzz, we include `sfilter` and `szip` as primitives. Fuzz does not include functions for clipping (specifically, `clipL1` and `clipL2`), because Fuzz's types cannot describe their sensitivity properties. We include `infsensL` to run a regular Haskell function on a sensitive list, which can be accomplished implicitly in Fuzz but requires a primitive in Solo. To support recursive functions over sensitive lists, we include `sfoldr`—which can be implemented as a recursive function in Fuzz, but not in Solo. We show how to

```
-- Doubles
sConstD  :: forall s. Double -> SDouble Diff s
(<+>)    :: SDouble m s1 -> SDouble m s2 -> SDouble m (s1 +++ s2)
(<->)    :: SDouble m s1 -> SDouble m s2 -> SDouble m (s1 +++ s2)
sabs     :: SDouble m s -> SDouble m s
-- Pairs & Lists
sConstL  :: forall s m t. [t s] -> SList m t s
scons    :: t s1 -> SList m t s2 -> SList m t (s1 +++ s2)
sfoldr   :: forall fn_sens1 fn_sens2 t1 t2 cm s3 s4 s5.
  (forall s1 s2. t1 s1 -> t2 s2 -> t2 ((ScaleSens s1 fn_sens1) +++ (ScaleSens s2 fn_sens2)))
  -> t2 s5 -> SList cm t1 s4 -> t2 ((ScaleSens s4 (MaxNat fn_sens1 fn_sens2)) +++ TruncateInf s5)
sfilter  :: (forall s. Double -> Bool) -> L1List (SDouble m) s1 -> L1List (SDouble m) s1
szip     :: SList m a s1 -> SList m b s2 -> SList m (L1Pair a b) (s1 +++ s2)
clipL1   :: forall m s. L1List (SDouble m) s -> L1List (SDouble Diff) (TruncateSens 1 s)
clipL2   :: forall m s. L2List (SDouble m) s -> L2List (SDouble Diff) (TruncateSens 1 s)
infsensL :: ([Double]->[Double]) -> SList cm (SDouble m) s -> SList cm (SDouble m) (TruncateInf s)
```

Fig. 2. Primitives in Solo

implement other recursive functions (including `map`) using `sfoldr` as a basic building block later in this section.

Sensitivity Identities. The primitives described so far allow us to implement useful programs, including recursive ones. For example, we can write a function that adds up the numbers in a sensitive list:

```
sumList xs = sfoldr (<+>) (sConstD '[] 0) xs
```

However, this definition results in a type error:

```
  Couldn't match type 's1 +++ s2' with 'ScaleSens s1 fn_sens10 +++ ScaleSens s2 fn_sens20'
```

The reason is that `sfoldr` expects the folded function to have an output type that includes **ScaleSens**, but `<+>` does not have this type. However, it is clear that the two types are equivalent: `<+>` is 1-sensitive in both arguments, which is equivalent to scaling both sensitivity environments by 1.

The Solo library includes several identities, described in Figure 3, for solving problems like this one. The programmer can apply these identities to resolve type errors where the two types are actually equivalent based on the properties of sensitivities. In this case, we want the `scale_unit` identity: a sensitivity environment by itself is equivalent to the same environment scaled by 1. We include a function `cong` (for congruence) to apply these identities, and `eq_sym` for symmetry; the full implementation appears in the supplemental materials [Abuah et al. 2021a] . We can address the type error in our example by modifying the function definition to be:

```
sumList :: L1List (SDouble Diff) s -> SDouble Diff s
sumList xs = cong scale_unit $ sfoldr @1 @1 scalePlus (sConstD @'[] 0) xs
  where scalePlus a b = cong (eq_sym scale_unit) a <+> cong (eq_sym scale_unit) b
```

Here, the definition of `scalePlus` has an output type with the required scaling, via the `scale_unit` identity; we also use `scale_unit` on the result of `sfoldr` to remove the scaling from the output. We also need to use type-level application to tell `sfoldr` that `scalePlus` is 1-sensitive in both arguments.

The manual application of these identities is an additional annotation burden not required in systems like Duet or DFuzz [de Amorim et al. 2015], whose implementations typically contain SMT or purpose-built solvers to solve the associated inference tasks. The set of identities in Figure 3 is unlikely to be exhaustive, and additional identitiest may need to be added—especially if more

```
    scale_unit:                  ScaleSens s 1   =  s
  maxnat_idemp:                    MaxNat n n    =  n
truncate_n_inf:  TruncateSens n (TruncateInf s)  =  TruncateSens n s
 scale_distrib:        ScaleSens (s1 +++ s2) n   =  ScaleSens s1 n +++ ScaleSens s2 n
 trunc_distrib:    TruncateSens (n1 TL.+ n2) s   =  TruncateSens n1 s +++
                                                       TruncateSens n2 s
   scale_cong1:                 ScaleSens s1 n   =  ScaleSens s2 n
   scale_cong2:                 ScaleSens s n1   =  ScaleSens s n2
```

Fig. 3. Sensitivity Identities

primitives are added to Solo. These identities must be implemented carefully, because Solo's soundness depends on their correctness.

Standard Library. Solo's standard library includes functions like `smap` and `count`, implemented using Solo's primitives and identities and verified automatically.

Map. The `map` function is implemented as a recursive function in Fuzz, and not provided in DPella. In Solo, it can be implemented using `scons` and `sfoldr`. We use the `scale_unit` axiom to show that the folded function is 1-sensitive.

```
smap :: forall fn_sens a b s2 m. (forall s1. a s1 -> b (ScaleSens s1 fn_sens))
  -> SList m a s2 -> SList m b (ScaleSens s2 (MaxNat fn_sens 1))
smap f as = sfoldr @fn_sens @1 (\x xs -> scons (f x) (cong (eq_sym scale_unit) xs))
                  (sConstL @'[] []) as
```

We also define `stmap`, which expects the mapped function to *truncate* the sensitivity environment, rather than scale it (this occurs when the mapped function performs clipping).

Count. The `count` function counts the number of elements of a sensitive list; it is implemented in Fuzz as a recursive function, and provided as a primitive in DPella. We define `count` using `fold`; we implement the helper `count_fn` in a **`where`** clause in order to quantify over the sensitivity environment `s1`. We use the `scale_unit` axiom in `count_fn` to show the folded function is 1-sensitive, and we use it in `count` to convert the explicit 1-sensitivity guarantee to the simpler type given in the definition.

```
count :: SList cm t s -> SDouble Diff s
count xs = cong scale_unit $ sfoldr @1 @1 count_fn (sConstD @'[] 0) xs
  where count_fn :: forall (s1 :: SEnv) s2 t.
          t s1 -> SDouble Diff s2 -> SDouble Diff (ScaleSens s1 1 +++ ScaleSens s2 1)
        count_fn x a = (cong (eq_sym scale_unit) $ sConstD @s1 1) <+> (cong (eq_sym scale_unit) a)
```

List summation. The `sListSum` function adds two lists together, and also adds their sensitivity environments. The output type is designed (using the `scale_distrib` identity) so that `sListSum` is easy to use with `sfoldr` (both sensitivity environments are scaled by 1).

```
sListSum :: forall s1 s2 m. SList m (SDouble Diff) s1 -> SList m (SDouble Diff) s2
  -> SList m (SDouble Diff) (ScaleSens s1 1 +++ ScaleSens s2 1)
sListSum xs ys = cong (scale_distrib @1 @s1 @s2) $ smap @1 pairPlus $ szip xs ys
```

Convenience for 1-sensitive functions. The `smap` and `sfold` functions are commonly used with 1-sensitive functions. In this case,the output sensitivity environment must have the form `ScaleSens s 1`. However, the target function often has the simpler (equivalent) annotation `s`, so we need to use the `scale_unit` identity before applying `smap`. For convenience in this situation, the

Solo standard library contains versions of common combinators specialized to this case, with the suffix `1s` (for 1-sensitive). The required identities are applied in the convenience function, so the programmer does not need to use them directly. For example, the 1-sensitive version of `sfold` is:

```
sfoldr1s :: forall t1 t2 cm s3 s4 s5. (forall s1 s2. t1 s1 -> t2 s2 -> t2 (s1 +++ s2))
        -> t2 s5 -> SList cm t1 s4 -> t2 (s4 +++ TruncateInf s5)
sfoldr1s f init xs = cong prf $ sfoldr @1 @1 f' init xs
  where f' :: t1 s1 -> t2 s2 -> t2 (ScaleSens s1 1 +++ ScaleSens s2 1)
        f' a b = f (cong (eq_sym scale_unit) a) (cong (eq_sym scale_unit) b)
        prf = plus_cong @(ScaleSens s4 1) @s4 @(TruncateInf s5) scale_unit Id
```

8 CASE STUDIES

This section presents three case studies in Solo, to validate its applicability to real differentially private algorithms and compare the required annotation burden with purpose-built type systems for differential privacy. Each case study algorithm has been previously verified using specialized type systems, but ours is the first static approach embedded in a mainstream language with this capability.

8.1 Cumulative Distribution Function

Our first case study implements the private `cdf` function [McSherry and Mahajan 2010] as verified in DFuzz [Gaboardi et al. 2013] and DPella [Lobo-Vesga et al. 2020]. Given a database of numeric records, and a set of buckets associated with cutoff values, the `cdf` function privately partitions each record to its respective bucket. As in DFuzz, this case study demonstrates the ability of Solo to verify privacy costs which depend on a program input.

```
cdf :: forall eps iters s. (TL.KnownNat (MaxSens s), TL.KnownNat iters) =>
  [Double] -> L1List (SDouble Disc) s -> PM (ScalePriv (TruncatePriv eps Zero s) iters) [Double]
cdf buckets db = seqloop @iters (\i results -> do
                                    let c = count $ sfilter ((<) $ buckets !! i) db
                                    r <- laplace @eps c
                                    return (r : results)) []
```

We represent the set of buckets using a (non-sensitive) list of Doubles, and loop for a fixed number of iterations using `seqloop`. The private input dataset is represented using an L_1-sensitive list of doubles with the discrete metric. An implementation using `map` over the buckets is also possible (as in DFuzz), but requires vectors with statically-known lengths. The type of `cdf`'s output corresponds to a privacy cost of $(k\epsilon, 0)$-differential privacy for k iterations.

Annotation burden. Solo's analysis requires minimal additional annotations for simple algorithms like `cdf`. In this case, the programmer needs to specify the types and metrics for the input, and Haskell is able to infer the output type. The annotation burden for this example is similar to the burden in purpose-built languages like Fuzz and Duet.

8.2 MWEM

Our second case study is the Multiplicative Weights with Exponential Mechanism (MWEM) algorithm [Hardt et al. 2012], previously verified using the Jazz system [Toro et al. 2020]. The MWEM algorithm builds a synthetic dataset approximating the private input by iteratively improving the synthetic data until its answers match the real data on a target set of linear queries. We use *range queries*—queries that count the number of records within a specific range. We represent range queries as pairs (the lower and upper bounds of the range), and implement `evaluateDB` (to evaluate

queries on the sensitive real data, represented as a L_1 list of sensitive doubles with the discrete metric) and `evaluateSynth` (to evaluate queries on the synthetic data):

```
type RangeQuery = (Double, Double)

evaluateDB :: RangeQuery -> L1List (SDouble Disc) s -> SDouble Diff s
evaluateDB (l, u) db = count $ sfilter (\x -> l < x && u > x) db

evaluateSynth :: RangeQuery -> [Double] -> Double
evaluateSynth (l, u) syn_rep = fromIntegral $ length $ filter (\x -> l < x && u > x) syn_rep
```

The MWEM algorithm uses the exponential mechanism [McSherry and Talwar 2007] to select a query for which the synthetic data performs badly, then uses the Laplace mechanism to evaluate that query on the real data. For the first step, we define a score function that computes the absolute value of the difference between a query's result on the real data and its result on the synthetic data:

```
scoreFn :: forall s. [Double] -> RangeQuery -> L1List (SDouble Disc) s -> SDouble Diff s
scoreFn syn_rep q db =
  let true_answer = evaluateDB q db
      syn_answer  = evaluateSynth q syn_rep
  in sabs $ sConstD @'[] syn_answer <-> true_answer
```

Here, we use the `sConstD` function to build a sensitive double from a constant—and attach an empty sensitivity environment—so that we can use the `<->` function. Finally, we can define `mwem`:

```
1  mwem :: forall eps iters s.
2    (TL.KnownNat (MaxSens s), TL.KnownNat iters) =>
3    [Double] -> [RangeQuery] -> L1List (SDouble Disc) s
4    -> PM (ScalePriv ((TruncatePriv eps Zero s) ++++ (TruncatePriv eps Zero s)) iters) [Double]
5  mwem syn_rep qs db =
6    let mwemStep _ syn_rep = do
7          selected_q <- expMech @eps (scoreFn syn_rep) qs db
8          measurement <- laplace @eps (evaluateDB selected_q db)
9          return $ multiplicativeWeights syn_rep selected_q measurement
10   in seqloop @iters mwemStep syn_rep
```

The algorithm runs `mwemStep` a pre-defined number of times using the `seqloop` looping combinator. Each iteration of `mwemStep` selects a query (line 7) using the exponential mechanism, then evaluates the selected query on the real data (line 8) with the Laplace mechanism. Finally, the algorithm updates the synthetic data (line 9) with the *multiplicative weights update rule* [Hardt and Rothblum 2010]. The rule can be defined as a *regular Haskell function* (omitted for space) because none of its arguments are sensitive. The type of `mwem`'s output indicates that it satsifies $(2k\epsilon, 0)$-differential privacy, for k iterations.

Annotation burden. MWEM is a more complicated algorithm, and requires the programmer to understand more about Solo's analysis. In addition to describing input types and metrics, the programmer needs to mix sensitive and non-sensitive types (using `sConstD`) in the definition of the score function. These additional annotations would not be required in purpose-built languages like Fuzz and Duet.

8.3 Gradient Descent

Our third case study considers differentially private gradient descent, widely used in machine learning and previously verified in Adaptive Fuzz [Winograd-Cort et al. 2017] and Duet [Near et al. 2019]. Gradient descent iteratively updates the *weights*, or parameters, of a model to minimize the *loss* when the model is evaluated on the training data. To determine how to update the weights, the

algorithm calculates the gradient of the loss. We represent the model's weights as a list of doubles, and a single training example (i.e. a feature vector) as a list of doubles. We define a sensitive example using an L_2 list, and a dataset (i.e. the training data) as an L_1 list of sensitive examples. We assume a definition of the gradient calculation as a *regular Haskell function* that calculates the gradient with respect to the weights based on the weights and one training example. This function can be written by the programmer, or provided by a machine learning library (e.g. the `backPropagate` function provided by the Grenade library [Grenade 2020]).

```
type Weights = [Double]
type Example = [Double]
type SExample = L2List (SDouble Disc)
type SDataset senv = L1List SExample senv
gradient :: Weights -> Example -> Weights
```

The `gradient` function provides no sensitivity guarantee. To ensure the gradient has bounded sensitivity, we can clip its output using `clipL2`:

```
clippedGrad :: forall senv cm m.
  Weights -> SExample senv -> L2List (SDouble Diff) (TruncateSens 1 senv)
clippedGrad weights x =
  let g = infsensL (gradient weights) x         -- apply the infinitely-sensitive function
  in cong (truncate_n_inf @1 @senv) $ clipL2 g  -- clip the results and return
```

Here, we use `infsensL` to run the inifinitely-sensitive function `gradient`, then use `clipL2` to enforce bounded sensitivity. We use `truncate_n_inf` to make the output type easier to read. Finally, we are ready to define the gradient descent algorithm:

```
1 gradientDescent :: forall ϵ δ iterations s.
2   (TL.KnownNat iterations) =>
3   Weights -> SDataset s -> PM (ScalePriv (TruncatePriv ϵ δ s) iterations) Weights
4 gradientDescent weights xs =
5   let gradStep i weights =
6         let clippedGrads = stmap @1 (clippedGrad weights) xs
7             gradSum = sfoldr1s sListSum1s (sConstL @'[] []) clippedGrads
8         in gaussLN @ϵ @δ @1 @s gradSum
9   in seqloop @iterations gradStep weights
```

The `gradStep` function represents one iteration of the algorithm. It calculates the L_2 clipped gradient for *each* example in the dataset (line 6) using `stmap`, the truncation form of `smap`. Then, the algorithm sums up the gradients elementwise, using `sfoldr` and `sListSum` (line 7). Finally, the algorithm applies the Gaussian mechanism (line 8). We use the `seqloop` combinator to run many iterations of the algorithm; the output type of the function indicates that it satisfies $(k\epsilon, k\delta)$-differential privacy for k iterations.

Gradient descent often runs for many iterations, so it is important to use the tightest possible bounds on sequential composition. We can take advantage of advanced composition (Section 6.4) by replacing the `seqloop` combinator with `advloop`; the resulting output type is:

```
PM (AdvComp iterations δ (TruncatePriv ϵ δ s)) Weights
```

This type indicates that the algorithm satisfies (ϵ', δ')-differential privacy, where ϵ' and δ' are calculated according to the advanced composition theorem. Given concrete values for ϵ and δ, Solo can apply the `AdvComp` type-level function to calculate actual values for ϵ' and δ'.

Annotation burden. Gradient descent is the most complex algorithm we verify, and it requires significant new annotations to deal with the truncation of sensitivity environments resulting from

clipping. For example, `clippedGrad` uses the `truncate_n_inf` identity to ensure a sensitivity bound that does not involve scaling the environment to infinity. The manual application of identities to prove sensitivity bounds is not required in purpose-built languages like Fuzz and Duet, and represents additional annotation burden required in Solo.

9 FORMALISM

In Solo, we implement a novel static analysis library for function sensitivity and differential privacy. Our approach can be seen as a type-and-effect system, which may be embedded in statically typed functional languages with support for monads and type-level arithmetic.

Our formalism demonstrates the feasibility of sound sensitivity analysis without linear types, using the approach implemented in Solo—by embedding the approach in a simple statically-typed programming language. In particular, our formalism's typing rules for variables, function introduction, and application are standard, and the additional rules are just for sensitive datatypes. Our formalism does not does not model all aspects of Haskell's type system—including polymorphism—so our formalism is not a direct proof of soundness for the Solo library. Instead, it is intended to validate the fundamental concept underlying Solo's design.

Like the Solo library, our formalism includes both "raw" types that do not attach sensitivity environments, and "sensitive" types that do attach sensitivity environments. Haskell's regular list type corresponds to our formalism's raw list type, and Solo's `SList` type corresponds to the sensitive list type in the formalism.

Program Syntax. Figure 4 shows a core subset of the syntax for our analysis system. Our language model includes arithmetic operations ($e \odot e$), pairs ($\langle e, e \rangle$ and $\pi_i(e)$), conditionals ($\texttt{if0}(e)\{e\}\{e\}$), and functions ($\lambda_x x.\ e$ and $e(e)$). Types τ presented in the formalism include: base numeric types $\texttt{real}$, singleton numeric types with a known runtime value at compile-time $\texttt{real}[r]$, booleans $\texttt{bool}$, functions $\tau \to \tau$, pairs $\tau \times \tau$, and the privacy monad $\bigcirc_\Sigma(\tau)$. Regular types τ are accompanied by sensitive types σ which are essentially regular types annotated with static sensitivity analysis information Σ—which is the sensitivity analysis (or sensitivity environment) for the expression which was typed as τ. Sensitive types shown in our formalism include sensitive numeric types $\texttt{sreal}$, sensitive pairs $\sigma \otimes \sigma$, and sensitive lists $\texttt{slist}(\sigma)$. A metric-carrying singleton numeric type is unnecessary since its value is fixed and cannot vary. Σ—the *sensitivity/privacy environment*—is defined as a mapping from sensitive sources $o \in \text{source}$ to scalar values which represent the sensitivity/privacy of the resulting value with respect to that source.

Types/values with standard treatment are not shown in our formalism, but included in our implementation with both regular and metric-carrying versions, include vectors and matrices which have known dimensions at compile-time via singleton natural number indices. Single natural numbers are also used to execute loops with statically known number of iterations and to help contruct sensitivity and privacy quantities.

Typing Rules. Figure 5 shows typing rules (select rules only, see supplemental material for full set [Abuah et al. 2021a]) in our system used to reason about the sensitivity of computations. The majority of these rules are standard, and modeled on the corresponding rules in our implementation language (Haskell). In particular, the rule for function introduction (T-LAM) does not mention sensitivities or sensitive types.

The rules with a shaded background ( ) are unique to Solo, and model the primitives described earlier in the paper. For example, the rule T-SPLUS models the addition operator, which adds the sensitivity environments attached to its arguments ($\Sigma_1 + \Sigma_2$). Addition of sensitivity environments is identical to Fuzz [Reed and Pierce 2010] and Duet [Near et al. 2019], and models the implementation

$b \in \mathbb{B}$ $r \in \mathbb{R}$ $\dot{r} \in \dot{\mathbb{R}} ::= r \mid \infty$ $x, z \in \mathsf{var}$ $o \in \mathsf{source}$

$\Sigma \in \mathsf{spenv}$	$\triangleq$	$\mathsf{source} \rightharpoonup \dot{\mathbb{R}}$	sensitivity/privacy environment
$\tau \in \mathsf{type}$	$::=$	$\mathtt{bool} \mid \mathtt{real} \mid \mathtt{real}[r]$	base and singleton types
	$\mid$	$\tau \times \tau \mid \mathtt{list}(\tau) \mid \tau \to \tau$	connectives
	$\mid$	$\bigcirc_\Sigma(\tau) \mid \sigma@\Sigma$	privacy monad and sensitive types
$\sigma \in \mathsf{stype}$	$::=$	$\mathtt{sreal} \mid \sigma \otimes \sigma \mid \mathtt{slist}(\sigma)$	sensitive types
$\odot \in \mathsf{binop}$	$::=$	$+ \mid \times \mid \rtimes$	operations
$e \in \mathsf{expr}$	$::=$	$x \mid b \mid r \mid \mathtt{sing}(r)$	variables and literals
	$\mid$	$e \odot e \mid \mathtt{if}(e)\{e\}\{e\}$	binary operations and conditionals
	$\mid$	$\langle e, e \rangle \mid \pi_i(e)$	pair creation and access
	$\mid$	$[] \mid e :: e$	list creation
	$\mid$	$\mathtt{case}(e)\{[].e\}\{x :: x.e\}$	list destruction
	$\mid$	$\lambda_x x.\ e \mid e(e)$	recursive functions
	$\mid$	$\mathtt{reveal}(e) \mid \mathtt{laplace}[e, e](e)$	privacy operations
	$\mid$	$\mathtt{return}(e) \mid x \leftarrow e\ ;\ e$	privacy monad
	$\mid$	$\hat{\langle} e, e \hat{\rangle} \mid \hat{\pi}_i(e)$	sensitive pair creation and access
	$\mid$	$\hat{[]} \mid e \mathbin{\hat{::}} e$	sensitive list creation
	$\mid$	$\mathtt{case}(e)\{\hat{[]}.e\}\{x \mathbin{\hat{::}} x.e\}$	sensitive list destruction
$\gamma \in \mathsf{venv}$	$\triangleq$	$\mathsf{var} \rightharpoonup \mathsf{value}$	evaluation environment
$\rho \in \mathsf{ddist}$	$\triangleq$	$\left\{ f \in \mathsf{value} \to \mathbb{R} \mid \sum_v f(v) = 1 \right\}$	discrete distributions (PMF)
$v \in \mathsf{value}$	$::=$	$b \mid r$	literals
	$\mid$	$\langle v, v \rangle$	pairs
	$\mid$	$[] \mid v :: v$	lists
	$\mid$	$\langle \lambda_x x.\ e \mid \gamma \rangle$	recursive closures
	$\mid$	ρ	distributions of values

Fig. 4. Syntax for types, expressions and values. = sensitivity sources, types and expressions unique to Solo.

described earlier:

$$(\Sigma_1 + \Sigma_2)(o) \triangleq \begin{cases} \Sigma_1(o) + \Sigma_2(o) & \text{if } o \in \Sigma_1 \text{ and } o \in \Sigma_2 \\ \Sigma_1(o) & \text{if } o \in \Sigma_1 \text{ but } o \notin \Sigma_2 \\ \Sigma_2(o) & \text{if } o \in \Sigma_2 \text{ but } o \notin \Sigma_1 \end{cases}$$

The pointwise maximum of two sensitivity environments ($\Sigma_1 \sqcup \Sigma_2$) is defined analogously, but with the numeric maximum instead of addition; it is used in the rule T-SPAIR for pairs. The rule T-STIMES describes multiplication of a sensitive value by a statically-known number, which *scales* the associated sensitivity environment. Sensitivity environment scaling $\mathtt{s}(\Sigma)$ is defined as $\mathtt{s}(\Sigma)(o) \triangleq \mathtt{s}(\Sigma(o))$. The truncation operation $\rceil \Sigma \lceil^\epsilon$ is also defined as seen in prior work [Near et al. 2019]. This operation converts sensitivity environments to privacy environments, by replacing each sensitivity in the environment with a consistent privacy cost (i.e. pointwise). While typing rules for arithmetic operations vary for the several permutations of static(singleton)/dynamic arguments, we only show the interesting cases for the multiplication operator. T-RETURN, T-REVEAL, and T-LAPLACE model the privacy primitives described in Section 6. T-BIND encodes Theorem 2.1 (sequential composition). In general, the typing rules are similar to previous work, except that the sensitivity and privacy environments are properties of (and embedded in) the types themselves, rather than being a property of the program context.

$$\Gamma \in \text{tenv} \triangleq \text{var} \rightharpoonup \text{type} \qquad \rceil \Sigma \lceil^s(o) \triangleq \rceil \Sigma(o) \lceil^s \qquad \rceil s \lceil^{s'} \triangleq \begin{cases} 0 & \text{if} \quad s \triangleq 0 \\ s' & \text{if} \quad s \neq 0 \end{cases}$$

$$\mathcal{R}(\texttt{sreal}) \triangleq \texttt{real} \qquad \mathcal{R}(\sigma \otimes \sigma) \triangleq \mathcal{R}(\sigma) \times \mathcal{R}(\sigma) \qquad \mathcal{R}(\texttt{slist}(\sigma)) \triangleq \texttt{list}(\mathcal{R}(\sigma))$$

$\boxed{\Gamma \vdash e : \tau}$

T-VAR
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

T-SING
$$\frac{}{\Gamma \vdash \texttt{sing}(r) : \texttt{real}[r]}$$

T-LAM
$$\frac{\{x \mapsto \tau_1, z \mapsto \tau_1 \to \tau_2\} \uplus \Gamma \vdash e : \tau_2}{\Gamma \vdash \lambda_z x.\ e : \tau_1 \to \tau_2}$$

T-APP
$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1(e_2) : \tau_2}$$

T-REVEAL
$$\frac{\Gamma \vdash e : \sigma@\Sigma}{\Gamma \vdash \texttt{reveal}(e) : \bigcirc_{\rceil \Sigma \lceil^\infty}(\mathcal{R}(\sigma))}$$

T-LAPLACE
$$\frac{\Gamma \vdash e_1 : \texttt{real}[r_s] \qquad \Gamma \vdash e_2 : \texttt{real}[r_\epsilon] \qquad \Gamma \vdash e_3 : \texttt{sreal}@\Sigma \qquad \Sigma \sqsubseteq \rceil \Sigma \lceil^s}{\Gamma \vdash \texttt{laplace}[e_1, e_2](e_3) : \bigcirc_{\rceil \Sigma \lceil^\epsilon}(\texttt{real})}$$

T-RETURN
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}(e) : \bigcirc_{\varnothing}(\tau)}$$

T-BIND
$$\frac{\Gamma \vdash e_1 : \bigcirc_{\Sigma_1}(\tau_1) \qquad \{x \mapsto \tau_1\} \uplus \Gamma \vdash e_2 : \bigcirc_{\Sigma_2}(\tau_2)}{\Gamma \vdash x \leftarrow e_1\ ;\ e_2 : \bigcirc_{\Sigma_1 + \Sigma_2}(\tau_2)}$$

T-SPLUS
$$\frac{\Gamma \vdash e_1 : \texttt{sreal}@\Sigma_1 \qquad \Gamma \vdash e_2 : \texttt{sreal}@\Sigma_2}{\Gamma \vdash e_1 + e_2 : \texttt{sreal}@(\Sigma_1 + \Sigma_2)}$$

T-STIMES
$$\frac{\Gamma \vdash e_1 : \texttt{real}[r] \qquad \Gamma \vdash e_2 : \texttt{sreal}@\Sigma}{\Gamma \vdash e_1 \ltimes e_2 : \texttt{sreal}@r\Sigma}$$

T-SPAIR
$$\frac{\Gamma \vdash e_1 : \sigma_1@\Sigma_1 \qquad \Gamma \vdash e_2 : \sigma_2@\Sigma_2}{\Gamma \vdash \hat{\langle} e_1, e_2 \hat{\rangle} : (\sigma_1 \otimes \sigma_2)@(\Sigma_1 \sqcup \Sigma_2)}$$

T-SPROJ
$$\frac{\Gamma \vdash e : (\sigma_1 \otimes \sigma_2)@\Sigma}{\Gamma \vdash \hat{\pi}_i(e) : \sigma_i@\Sigma}$$

T-SNIL
$$\frac{}{\Gamma \vdash \hat{[}\hat{]} : \texttt{slist}(\sigma)@\varnothing}$$

T-SCONS
$$\frac{\Gamma \vdash e_1 : \sigma@\Sigma_1 \qquad \Gamma \vdash e_2 : \texttt{slist}(\sigma)@\Sigma_2}{\Gamma \vdash e_1 \hat{::} e_2 : \texttt{slist}(\tau)@(\Sigma_1 \sqcup \Sigma_2)}$$

T-SCASE
$$\frac{\Gamma \vdash e_1 : \texttt{slist}(\sigma)@\Sigma \qquad \Gamma \vdash e_2 : \tau' \qquad \{x_1 \mapsto \sigma@\Sigma, x_2 \mapsto \texttt{slist}(\sigma)@\Sigma\} \uplus \Gamma \vdash e_3 : \tau'}{\Gamma \vdash \texttt{case}(e_1)\{\hat{[}\hat{]}.e_2\}\{x_1 \hat{::} x_2.e_3\} : \tau'}$$

$$\bar{\rho} \in \text{value} \to \text{ddist} \qquad \bar{n} \in \text{value} \to \mathbb{N}$$

$\boxed{\gamma \vdash e \Downarrow_n v}$

E-VAR
$$\frac{\gamma(x) = v}{\gamma \vdash x \Downarrow_0 v}$$

E-PLUS
$$\frac{\gamma \vdash e_1 \Downarrow_{n_1} r_1 \qquad \gamma \vdash e_2 \Downarrow_{n_2} r_2}{\gamma \vdash e_1 + e_2 \Downarrow_{n_1+n_1} r_1 + r_2}$$

E-LAM
$$\frac{}{\gamma \vdash \lambda_z x.\ e \Downarrow_0 \langle \lambda_z x.\ e \mid \gamma \rangle}$$

E-APP
$$\frac{\gamma \vdash e_1 \Downarrow_{n_1} \langle \lambda_z x.\ e' \mid \gamma' \rangle \qquad \gamma \vdash e_2 \Downarrow_{n_2} v_1 \qquad \{x \mapsto v_1, z \mapsto \langle \lambda_z x.\ e' \mid \gamma' \rangle\} \uplus \gamma' \vdash e' \Downarrow_{n_3} v_2}{\gamma \vdash e_1(e_2) \Downarrow_{n_1+n_2+n_3+1} v_2}$$

E-REVEAL
$$\frac{\gamma \vdash e \Downarrow_n v}{\gamma \vdash \texttt{reveal}(e) \Downarrow_n \{v \mapsto 1\}} \qquad \gamma \vdash \texttt{return}(e) \Downarrow_n \{v \mapsto 1\}$$

E-LAPLACE
$$\frac{\gamma \vdash e_1 \Downarrow_n s \qquad \gamma \vdash e_2 \Downarrow_n \epsilon \qquad \gamma \vdash e_3 \Downarrow_n r}{\gamma \vdash \texttt{laplace}[e_1, e_2](e_3) \Downarrow_n \text{laplace}(r, s/\epsilon)}$$

E-BIND
$$\frac{\gamma \vdash e_1 \Downarrow_{n_1} \rho_1 \qquad \forall v.\ \{x \mapsto v\} \uplus \gamma \vdash e_2 \Downarrow_{\bar{n}_2(v)} \bar{\rho}_2(v)}{\gamma \vdash x \leftarrow e_1\ ;\ e_2 \Downarrow_{\left(n_1 + \bigsqcup_v \bar{n}_2(v)\right)} \left\{ v \mapsto \sum_{v'} \rho_1(v') \bar{\rho}_2(v')(v) \right\}}$$

Fig. 5. Type system and step-indexed evaluation semantics, select rules only. ■ = type rules unique to Solo.

Dynamic Semantics. Figure 5 shows the dynamic semantics (select rules only, see supplemental material for full set [Abuah et al. 2021a]) that accompanies the syntax for our analysis system. Our semantics largely follows the structure of Fuzz [Reed and Pierce 2010], and model the evaluation of expressions to *discrete distributions* of values (since our privacy mechanisms are randomized). Distributions are represented as mappings from values to their probabilities (i.e. probability mass functions). The rule E-REVEAL says that a deterministic reveal of a value produces a point distribution

($\{v \mapsto 1\}$); the rule E-BIND encodes sequential composition. The rule E-LAPLACE returns a discrete Laplace distribution centered at r with scale s/ϵ.

Type Soundness. The property of type soundness in our system is defined (as in prior work) as the *metric preservation* theorem. Essentially, metric preservation dictates a maximum variation which is possible when a sensitive open term is closed over by two distinct but related sensitive closure environments. This means that given related initial well-typed configurations, we expect the outputs to be related by some level of variation. Specifically: given two well-typed environments which are related by the logical relation (values may be apart by distance Σ, for n steps), and a well typed term, then each evaluation of that term in each environment is related by the relation, that is, when one side terminates in $< n$ steps to a value, the other side will deterministically terminate to a related value. Similar to prior work, in order to state and prove the metric preservation theorem, we define the notion of function sensitivity as a (step-indexed) logical relation. Figure 6 shows the step-indexed logical relation (select rules only, see supplemental material for full set [Abuah et al. 2021a]) used to define function sensitivity. We briefly describe the logical relations seen in this figure, then state the metric preservation theorem formally.

(1) Two real numbers are related $r_1 \sim^r r_2$ at type $\mathbb{R}$ and distance r when the absolute difference between real numbers r_1 and r_2 is less than r.
(2) Two values are related $v_1 \sim v_2$ in $\mathcal{V}_\Sigma[\![\tau]\!]$ when v_1 and v_2 are related at type τ for initial distance Σ. We may define relatedness for the syntactic category of values via case analysis as follows:
   (a) Base numeric values are related $r_1 \sim^\Sigma r_2$ at type $\mathbb{R}$ in $\mathcal{V}_{\Sigma_1}[\![\tau]\!]$ when r_1 and r_2 are related by $\Sigma \cdot \Sigma_1$, where Σ is the initial distances between each input source o, and Σ_1 describes how much these values may wiggle as function arguments i.e. the maximum permitted argument variation. $\cdot$ is defined as the vector dot product.
   (b) Function values $\langle \lambda x.\ e_1 \mid \gamma_1 \rangle \sim \langle \lambda x.\ e_2 \mid \gamma_2 \rangle$ are related at type $(\tau \to \tau)$ in $\mathcal{V}_\Sigma[\![\tau]\!]$ when given related inputs, they produce related computations.
   (c) Pair values $\langle v_{11}, v_{12} \rangle \sim \langle v_{21}, v_{22} \rangle$ are related at type $\langle \tau, \tau \rangle$ in $\mathcal{V}_\Sigma[\![\tau]\!]$ when they are element-wise related.
   (d) $\gamma_1, e_1 \sim \gamma_2, e_2$ are related at type τ and distance Σ in $\mathcal{E}_\Sigma[\![\tau]\!]$ when the input doubles γ_1, e_1 and γ_2, e_2 evaluate to output values which are related by Σ.
(3) Two value environments $\gamma_1 \sim \gamma_2$ are related at type environment Γ and sensitivity environment Σ in $\mathcal{G}_\Sigma[\![\Gamma]\!]$ if value environments γ_1 and γ_2 both map each variable in the type environment Γ to related values at a matching type at distance Σ.

Theorem 9.1 (Metric Preservation). *If $\gamma_1 \sim \gamma_2 \in \mathcal{G}_n^\Sigma[\![\Gamma]\!]$ and $\Gamma \vdash e : \tau$, then $\gamma_1, e \sim \gamma_2, e \in \mathcal{E}_n^\Sigma[\![\tau]\!]$. That is, either $n = 0$, or $n = n' + 1$ and if $n'' \leq n'$ and $\gamma_1 \vdash e \Downarrow_{n''} v_1$, then $\exists! v_2.\ \gamma_2 \vdash e \Downarrow_{n''} v_2$ and $v_1 \sim v_2 \in \mathcal{V}_{n'-n''}^\Sigma[\![\tau]\!]$.*

The proofs appear in the supplemental material [Abuah et al. 2021a] .

10 RELATED WORK

Lightweight Static Analysis for Differential Privacy. The DPella [Lobo-Vesga et al. 2020] system is closest to our work. Like Solo, DPella uses Haskell's type system for sensitivity analysis, but DPella implements a custom dynamic analysis of programs to compute privacy and accuracy information. Solo goes beyond DPella by supporting calculation of privacy costs using Haskell's type system, in addition to sensitivity information.

Linear Types. Fuzz was the first language and type system designed to verify differential privacy costs of a program, and did so by modeling sensitivity using linear types [Reed and Pierce 2010]. DFuzz extended Fuzz with dependent types and automation aided by SMT solvers [Gaboardi et al.

$$\gamma_1, e_1 \sim \gamma_2, e_2 \in \mathcal{E}^{\Sigma}_n[\![\tau]\!] \overset{\Delta}{\iff} n = 0 \implies \text{true}$$
$$\wedge\ n = n' + 1 \implies \forall n'' \le n', v_1.\ \gamma_1 \vdash e_1 \Downarrow_{n''} v_1$$
$$\implies \exists! v_2.\ \gamma_2 \vdash e_2 \Downarrow_{n''} v_2 \ \wedge\ v_1 \sim v_2 \in \mathcal{V}^{\Sigma}_{n'-n''}[\![\tau]\!]$$

$\boxed{\gamma, e \sim \gamma, e \in \mathcal{E}^{\Sigma}_n[\![\tau]\!]}$

$$r_1 \sim r_2 \in \mathcal{V}^{\Sigma}_n[\![\texttt{sreal@}\Sigma']\!] \overset{\Delta}{\iff} |r_1 - r_2| \le \Sigma{\cdot}\Sigma'$$

$\boxed{v \sim v \in \mathcal{V}^{\Sigma}_n[\![\tau]\!]}$

$$\langle v_{11}, v_{12}\rangle \sim \langle v_{21}, v_{22}\rangle \in \mathcal{V}^{\Sigma}_n[\![(\sigma_1 \otimes \sigma_2)@\Sigma']\!] \overset{\Delta}{\iff} v_{11} \sim v_{21} \in \mathcal{V}^{\Sigma}_n[\![\sigma_1@\Sigma']\!]$$
$$\wedge\ v_{12} \sim v_{22} \in \mathcal{V}^{\Sigma}_n[\![\sigma_2@\Sigma']\!]$$

$$v_{11} \mathbin{\hat{::}} v_{12} \sim v_{21} \mathbin{\hat{::}} v_{22} \in \mathcal{V}^{\Sigma}_n[\![\texttt{slist}(\sigma)@\Sigma']\!] \overset{\Delta}{\iff} v_{11} \sim v_{21} \in \mathcal{V}^{\Sigma}_n[\![\sigma@\Sigma']\!]$$
$$\wedge\ v_{12} \sim v_{22} \in \mathcal{V}^{\Sigma}_n[\![\texttt{slist}(\sigma)@\Sigma']\!]$$

$$\langle \lambda_z x.e_1 | \gamma_1\rangle \sim \langle \lambda_z x.e_2 | \gamma_2\rangle \in \mathcal{V}^{\Sigma}_n[\![\tau_1 \to \tau_2]\!] \overset{\Delta}{\iff} \forall n' \le n, v_1, v_2.\ v_1 \sim v_2 \in \mathcal{V}^{\Sigma}_{n'}[\![\tau_1]\!]$$
$$\implies \{x \mapsto v_1, z \mapsto \langle \lambda_z x.e_1 \mid \gamma_1\rangle\} \uplus \gamma_1, e_1$$
$$\sim \{x \mapsto v_2, z \mapsto \langle \lambda_z x.e_2 \mid \gamma_2\rangle\} \uplus \gamma_2, e_2$$
$$\in \mathcal{E}^{\Sigma}_{n'}[\![\tau_2]\!]$$

$$\rho_1 \sim \rho_2 \in \mathcal{V}^{\Sigma}_n[\![\bigcirc_{\Sigma'}(\tau)]\!] \overset{\Delta}{\iff} \forall v.\ \rho_1(v) \le e^{|\lceil \Sigma \rceil^{\dagger} \times \Sigma'|_{L^\infty}} \rho_2(v)$$

Fig. 6. Step-indexed logical relation, select rules only.

2013]. The Duet language extends Fuzz with support for advanced variants of differential privacy such as (ϵ, δ)-differential privacy [Near et al. 2019]. Adaptive Fuzz embeds a static sensitivity analysis within a dynamic privacy analysis using privacy odometers and filters [Winograd-Cort et al. 2017]. The above approaches all require linear types, which are typically not available in mainstream programming languages. The Granule language [Orchard et al. 2019] is specifically designed to support linear types, but has not yet been widely adopted by programmers.

Indexed Monadic Types. Azevedo de Amorim et al [de Amorim et al. 2018] introduce a path construction to embed relational tracking for (ϵ, δ)-differential privacy within the Fuzz type system. This technique internalizes *group privacy* and can produce non-optimal privacy bounds.

Program Logics, Randomness Alignments, & Probabilistic Couplings Program logics such as apRHL [Barthe et al. 2012, 2013] are very flexible and expressive but difficult to automate. Fuzzi [Zhang et al. 2019] combines the Fuzz type system (for composition of sensitivity and privacy operations) with apRHL (for proofs of basic mechanisms) to eliminate the need for trusted primitives like the Laplace mechanism. Approaches based on randomness alignments, such as LightDP [Zhang and Kifer 2017] and ShadowDP [Wang et al. 2019] are suitable for verifying low level techniques such as the sparse vector technique [Dwork et al. 2014b] but not for sensitivity analysis. Barthe et al introduce an approach for proving differential privacy using a generalization of probabilistic couplings. They present several case studies in the apRHL$^+$ [Barthe et al. 2016] language which extends program logics with approximate couplings. The technique of aligning randomness is also used in the coupling method. Albarghouthi and Hsu [Albarghouthi and Hsu 2018] use an alternative approach based on randomness alignments as well as approximate couplings. None of these approaches can be easily embedded in mainstream languages like Haskell.

Dynamic Analyses. PINQ [McSherry 2009] pioneered dynamic enforcement of differential privacy for a subset of relational database query tasks. Featherweight PINQ [Ebadi and Sands 2015] is a framework which models PINQ and presents a proof that any programs which use its API are differentially private. ProPer [Ebadi et al. 2015] is also based on PINQ, but is primarily designed to maintain a privacy budget for each individual in a database system. ProPer operates by silently dropping records from queries when their privacy budget is exceeded. UniTrax [Munz et al. 2018] improves on ProPer by allowing per-user budgets without silently dropping records. UniTrax

operates by tracking queries against an abstract database as opposed to the actual database records. Diffprivlib [Holohan et al. 2019] (for Python) and Google's library [Wilson et al. 2020] (for several languages) provide differentially private algorithms for modern machine-learning and general data analysis. ϵktelo [Zhang et al. 2018] describes differentially private programs as *plans* over high level libraries of *operators* which have classes for data transformation, reduction, inference and other tasks. DDuo [Abuah et al. 2021b] extends PINQ-style dynamic analysis to general-purpose Python programs. Dynamic approaches require running the program in order to verify differential privacy, and in many cases add significant runtime overhead.

Dynamic Testing. Recent work by Bichsel et al. [Bichsel et al. 2018], Ding et al. [Ding et al. 2018], Wang et al. [Wang et al. 2020], and Wilson et al. [Wilson et al. 2020] have given rise to a set of techniques which facilitate testing for differential privacy. These approaches work for arbitrary programs written in any language, but they typically involve evaluating a program many times on neighboring inputs to check for possible violations of differential privacy—which can be intractable for complex algorithms.

Static Taint Analysis and IFC. Li et al [Peng Li and Zdancewic 2006] present an embedded security sublanguage in Haskell using the arrows combinator interface. Russo et al introduce a monadic library for light-weight information flow security in Haskell [Russo et al. 2008]. Crockett et al propose a domain specific language for safe homomorphic encryption in Haskell [Crockett et al. 2018]. Safe Haskell [Terei et al. 2012] is a Haskell language extension which implements various security policies as monads. Parker et al [Parker et al. 2019] introduce a Haskell framework for enforcing information flow control policies in database-oriented web applications.

Solo's sensitivity tracking is similar to approaches for tracking information flow, but it is more quantitative and follows a probabilistic programming structure (e.g. sampling from distributions). Solo thus has the structure of a taint analysis, but is refined to capture the specific information flow property of differential privacy. In particular, the sensitivity and privacy environments that we attach to the types of values can be seen as similar to IFC labels [Arzt et al. 2014; Buiras et al. 2015; Li et al. 2014; Myers 1999; Sridharan et al. 2011; Tripp et al. 2009; Wang et al. 2008; Yang and Yang 2012]

11 CONCLUSION

We have presented Solo, a lightweight static analysis approach for differential privacy. Solo can be embedded in mainstream functional languages, without the need for a specialized type system. We have proved the soundness (metric preservation) of Solo using a logical relation to establish function sensitivity. We have presented several case studies verifying differentially private algorithms seen in related work.

ACKNOWLEDGMENTS

This material is based upon work supported by DARPA under Contract No. HR001120C0087, National Science Foundation (NSF) under award 1901278, and Department of Energy (DOE) under award DE-SC0022396. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA, NSF or DOE.

REFERENCES

John M. Abowd. 2018. The U.S. Census Bureau Adopts Differential Privacy. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (London, United Kingdom) *(KDD '18)*. Association for Computing Machinery, New York, NY, USA, 2867. https://doi.org/10.1145/3219819.3226070

Chiké Abuah, David Darais, and Joseph Near. 2022. *Paper Artifact: Solo: A Lightweight Static Analysis for Differential Privacy.* https://doi.org/10.5281/zenodo.7079930

Chike Abuah, David Darais, and Joseph P Near. 2021a. Solo: A Lightweight Static Analysis for Differential Privacy. *arXiv preprint arXiv:2105.01632* (2021).

Chike Abuah, Alex Silence, David Darais, and Joe Near. 2021b. DDUO: General-Purpose Dynamic Analysis for Differential Privacy. *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)* (2021).

Aws Albarghouthi and Justin Hsu. 2018. Synthesizing coupling proofs of differential privacy. *PACMPL* 2, POPL (2018), 58:1–58:30. https://doi.org/10.1145/3158146

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *SIGPLAN Not.* 49, 6 (June 2014), 259–269. https://doi.org/10.1145/2666356.2594299

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy. In *POPL*. ACM, 55–68.

Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 2016. Proving Differential Privacy via Probabilistic Couplings. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) *(LICS '16)*. Association for Computing Machinery, New York, NY, USA, 749–758. https://doi.org/10.1145/2933575.2934554

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. 2012. Probabilistic Relational Reasoning for Differential Privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Philadelphia, PA, USA) *(POPL '12)*. Association for Computing Machinery, New York, NY, USA, 97–110. https://doi.org/10.1145/2103656.2103670

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. 2013. Probabilistic Relational Reasoning for Differential Privacy. *ACM Trans. Program. Lang. Syst.* 35, 3, Article 9 (Nov. 2013), 49 pages. https://doi.org/10.1145/2492061

Benjamin Bichsel, Timon Gehr, Dana Drachsler-Cohen, Petar Tsankov, and Martin Vechev. 2018. Dp-finder: Finding differential privacy violations by sampling and optimization. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 508–524.

William J. Bowman and Amal Ahmed. 2015. Noninterference for Free. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 101–113. https://doi.org/10.1145/2784731.2784733

Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing Static and Dynamic Typing for Information-Flow Control in Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. Association for Computing Machinery, New York, NY, USA, 289–301. https://doi.org/10.1145/2784731.2784758

Mark Bun and Thomas Steinke. 2016. Concentrated differential privacy: Simplifications, extensions, and lower bounds. In *Theory of Cryptography Conference*. Springer, 635–658.

Eric Crockett, Chris Peikert, and Chad Sharp. 2018. ALCHEMY: A Language and Compiler for Homomorphic Encryption Made EasY. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 1020–1037. https://doi.org/10.1145/3243734.3243828

Arthur Azevedo de Amorim, Emilio Jesús Gallego Arias, Marco Gaboardi, and Justin Hsu. 2015. Really Natural Linear Indexed Type Checking. *CoRR* abs/1503.04522 (2015). arXiv:1503.04522 http://arxiv.org/abs/1503.04522

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. 2018. Metric Semantics for Probabilistic Relational Reasoning. *CoRR* abs/1807.05091 (2018). arXiv:1807.05091 http://arxiv.org/abs/1807.05091

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. 2019. Probabilistic Relational Reasoning via Metrics. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–19.

Zeyu Ding, Yuxin Wang, Guanhong Wang, Danfeng Zhang, and Daniel Kifer. 2018. Detecting violations of differential privacy. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 475–489.

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *Theory of cryptography conference*. Springer, 265–284.

Cynthia Dwork, Aaron Roth, et al. 2014a. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3–4 (2014), 211–407.

Cynthia Dwork, Aaron Roth, et al. 2014b. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3–4 (2014), 211–407.

Hamid Ebadi and David Sands. 2015. Featherweight PINQ. arXiv:1505.02642 [cs.PL]

Hamid Ebadi, David Sands, and Gerardo Schneider. 2015. Differential Privacy: Now it's Getting Personal. *ACM SIGPLAN Notices* 50. https://doi.org/10.1145/2676726.2677005

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. 2013. Linear dependent types for differential privacy. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 357–370.

Grenade. 2020. Grenade Machine Learning Library. https://github.com/HuwCampbell/grenade

Moritz Hardt, Katrina Ligett, and Frank McSherry. 2012. A simple and practical algorithm for differentially private data release. In *Advances in Neural Information Processing Systems*. 2339–2347.

Moritz Hardt and Guy N Rothblum. 2010. A multiplicative weights mechanism for privacy-preserving data analysis. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. IEEE, 61–70.

Naoise Holohan, Stefano Braghin, Pól Mac Aonghusa, and Killian Levacher. 2019. Diffprivlib: the IBM differential privacy library. *arXiv preprint arXiv:1907.02444* (2019).

Shin-ya Katsumata. 2014. Parametric Effect Monads and Semantics of Effect Systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. Association for Computing Machinery, New York, NY, USA, 633–645. https://doi.org/10.1145/2535838.2535846

Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick McDaniel. 2014. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. arXiv:1404.7431 [cs.SE]

Elisabet Lobo-Vesga, Alejandro Russo, and Marco Gaboardi. 2020. A Programming Framework for Differential Privacy with Accuracy Concentration Bounds. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 411–428.

Min Lyu, Dong Su, and Ninghui Li. 2017. Understanding the Sparse Vector Technique for Differential Privacy. *Proceedings of the VLDB Endowment* 10, 6 (2017).

Frank McSherry and Ratul Mahajan. 2010. Differentially-Private Network Trace Analysis. In *Proceedings of the ACM SIGCOMM 2010 Conference* (New Delhi, India) *(SIGCOMM '10)*. Association for Computing Machinery, New York, NY, USA, 123–134. https://doi.org/10.1145/1851182.1851199

Frank McSherry and Kunal Talwar. 2007. Mechanism design via differential privacy. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07)*. IEEE, 94–103.

Frank D. McSherry. 2009. Privacy Integrated Queries: An Extensible Platform for Privacy-Preserving Data Analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/1559845.1559850

Ilya Mironov. 2017. Rényi Differential Privacy. In *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*. IEEE Computer Society, 263–275. https://doi.org/10.1109/CSF.2017.11

Reinhard Munz, Fabienne Eigner, Matteo Maffei, Paul Francis, and Deepak Garg. 2018. UniTraX: Protecting Data Privacy with Discoverable Biases. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer International Publishing, Cham, 278–299.

Andrew C. Myers. 1999. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, USA) *(POPL '99)*. Association for Computing Machinery, New York, NY, USA, 228–241. https://doi.org/10.1145/292540.292561

Chaya Nayak. 2020. New privacy-protected Facebook data for independent research on social media's impact on democracy. https://research.fb.com/blog/2020/02/new-privacy-protected-facebook-data-for-independent-research-on-social-medias-impact-on-democracy/

Joseph P Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, et al. 2019. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. 2019. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–30.

Dominic Orchard and Tomas Petricek. 2014. Embedding Effect Systems in Haskell. *SIGPLAN Not.* 49, 12 (Sept. 2014), 13–24. https://doi.org/10.1145/2775050.2633368

D. Orchard, Tomas Petricek, and A. Mycroft. 2014. The semantic marriage of monads and effects. *ArXiv* abs/1401.5391 (2014).

James Parker, Niki Vazou, and Michael Hicks. 2019. LWeb: Information Flow Security for Multi-Tier Web Applications. *Proc. ACM Program. Lang.* 3, POPL, Article 75 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290388

Peng Li and S. Zdancewic. 2006. Encoding information flow in Haskell. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*. 12 pp.–16. https://doi.org/10.1109/CSFW.2006.13

Tomas Petricek. 2017. *Context-aware programming languages*. Ph.D. Dissertation. University of Cambridge.

Jason Reed and Benjamin C Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*. 157–168.

Alejandro Russo, Koen Claessen, and John Hughes. 2008. A Library for Light-Weight Information-Flow Security in Haskell. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell* (Victoria, BC, Canada) *(Haskell '08)*. Association for Computing Machinery, New York, NY, USA, 13–24. https://doi.org/10.1145/1411286.1411289

Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarnieri, Omer Tripp, and Ryan Berg. 2011. F4F: Taint Analysis of Framework-Based Web Applications. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Portland, Oregon, USA) *(OOPSLA '11)*. Association for Computing Machinery, New York, NY, USA, 1053–1068. https://doi.org/10.1145/2048066.2048145

David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe Haskell. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 137–148. https://doi.org/10.1145/2364506.2364524

Matías Toro, David Darais, Chike Abuah, Joe Near, Damián Árquez, Federico Olmedo, and Éric Tanter. 2020. Contextual Linear Types for Differential Privacy. *arXiv preprint arXiv:2010.11342* (2020).

Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. 2009. TAJ: Effective Taint Analysis of Web Applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Dublin, Ireland) *(PLDI '09)*. Association for Computing Machinery, New York, NY, USA, 87–97. https://doi.org/10.1145/1542476.1542486

Philip Wadler and Peter Thiemann. 2003. The Marriage of Effects and Monads. *ACM Trans. Comput. Logic* 4, 1 (Jan. 2003), 1–32. https://doi.org/10.1145/601775.601776

X. Wang, Y. Jhi, S. Zhu, and P. Liu. 2008. STILL: Exploit Code Detection via Static Taint and Initialization Analyses. In *2008 Annual Computer Security Applications Conference (ACSAC)*. 289–298. https://doi.org/10.1109/ACSAC.2008.37

Yuxin Wang, Zeyu Ding, Daniel Kifer, and Danfeng Zhang. 2020. CheckDP: An Automated and Integrated Approach for Proving Differential Privacy or Finding Precise Counterexamples. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 919–938.

Yuxin Wang, Zeyu Ding, Guanhong Wang, Daniel Kifer, and Danfeng Zhang. 2019. Proving differential privacy with shadow execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 655–669.

Royce J Wilson, Celia Yuxing Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. 2020. Differentially Private SQL with Bounded User Contribution. *Proceedings on Privacy Enhancing Technologies* 2020, 2 (2020).

Daniel Winograd-Cort, Andreas Haeberlen, Aaron Roth, and Benjamin C. Pierce. 2017. A framework for adaptive differential privacy. *Proc. ACM Program. Lang.* 1, ICFP (2017), 10:1–10:29. https://doi.org/10.1145/3110254

Z. Yang and M. Yang. 2012. LeakMiner: Detect Information Leakage on Android with Static Taint Analysis. In *2012 Third World Congress on Software Engineering*. 101–104. https://doi.org/10.1109/WCSE.2012.26

Danfeng Zhang and Daniel Kifer. 2017. LightDP: towards automating differential privacy proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 888–901.

Dan Zhang, Ryan McKenna, Ios Kotsogiannis, Michael Hay, Ashwin Machanavajjhala, and Gerome Miklau. 2018. Ektelo: A framework for defining differentially-private computations. In *Proceedings of the 2018 International Conference on Management of Data*. 115–130.

Hengchu Zhang, Edo Roth, Andreas Haeberlen, Benjamin C Pierce, and Aaron Roth. 2019. Fuzzi: A three-level logic for differential privacy. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–28.

Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *Proceedings of the 21st ACM Conference on Computer and Communications Security*. Scottsdale, Arizona. https://arxiv.org/abs/1407.6981