Bunched Fuzz: Sensitivity for Vector Metrics

june wunder[1](✉), Arthur Azevedo de Amorim[3], Patrick Baillot[2], and Marco Gaboardi[1]

[1] Boston University, Boston, USA
jwunder@bu.edu
[2] Univ. Lille, CNRS, Inria, Centrale Lille, UMR9189 CRIStAL, F-59000 Lille, France
[3] Rochester Institute of Technology, Rochester, USA

Abstract. *Program sensitivity* measures the distance between the outputs of a program when run on two related inputs. This notion, which plays a key role in areas such as data privacy and optimization, has been the focus of several program analysis techniques introduced in recent years. Among the most successful ones, we can highlight type systems inspired by linear logic, as pioneered by Reed and Pierce in the Fuzz programming language. In Fuzz, each type is equipped with its own distance, and sensitivity analysis boils down to type checking. In particular, Fuzz features two product types, corresponding to two different notions of distance: the *tensor product* combines the distances of each component by *adding* them, while the *with product* takes their *maximum*.

In this work, we show that these products can be generalized to arbitrary L^p *distances*, metrics that are often used in privacy and optimization. The original Fuzz products, tensor and with, correspond to the special cases L^1 and L^∞. To ease the handling of such products, we extend the Fuzz type system with *bunches*—as in the logic of bunched implications—where the distances of different groups of variables can be combined using different L^p distances. We show that our extension can be used to reason about quantitative properties of probabilistic programs.

1 Introduction

When developing a data-driven application, we often need to analyze its *sensitivity*, or *robustness*, a measure of how its outputs can be affected by varying its inputs. For example, to analyze the privacy guarantees of a program, we might consider what happens when we include the data of one individual in its inputs [11]. When analyzing the stability of a machine-learning algorithm, we might consider what happens when we modify one sample in the training set [7].

Such applications have spurred the development of several techniques to reason about program sensitivity [23,9]. One successful approach is based on linear-like [14] type systems, as pioneered in Reed and Pierce's *Fuzz* language [23].

The basic idea behind Fuzz is to use typing judgments to track the sensitivity of a program with respect to each variable. Each type comes equipped with a notion of distance, and the typing rules explain how to update variable sensitivities for each operation. Because different distances yield different sensitivity analyses, it is often useful to endow a set of values with different distances, which

T. Wies (Ed.): ESOP 2023, LNCS 13990, pp. 451–478, 2023.
https://doi.org/10.1007/978-3-031-30044-8_17

leads to different Fuzz types. For example, like linear logic, Fuzz has two notions of products: the tensor product $\otimes$ and the Cartesian product & (with). The first one is equipped with the L^1 (or Manhattan) distance, where the distance between two pairs is computed by *adding* the distances between the corresponding components. The second one is equipped with the L^∞ (or Chebyshev) distance, where the component distances are combined by taking their *maximum*.

The reason for focusing on these two product types is that they play a key role in differential privacy [11], a rigorous notion of privacy that was the motivating application behind the original Fuzz design. However, we could also consider equipping pairs with more general L^p *distances*, which interpolate between the L^1 and L^∞ and are extensively used in convex optimization [8], information theory [10] and statistics [15]. Indeed, other type systems for differential privacy inspired by Fuzz [20] include types for vectors and matrices under the L^2 distance, which are required to use the Gaussian mechanism, one of the popular building blocks of differential privacy. Supporting more general L^p metrics would allow us to capture even more such building blocks [17,1], which would enable further exploration of the tradeoffs between differential privacy and accuracy.

In this paper, we extend these approaches and show that Fuzz can be enriched with a family of tensor products $\otimes_p$, for $1 \leq p \leq \infty$. These tensor products are equipped with the L^p distance, the original Fuzz products $\otimes$ and & corresponding to the special cases $\otimes_1$ and $\otimes_\infty$. Moreover, each connective $\otimes_p$ is equipped with a corresponding "linear implication" $\multimap_p$, unlike previous related systems where such an implication only exists for $p = 1$. Following prior work [4,3], we give to our extension a semantics in terms of non-expansive functions, except that the presence of the implications $\multimap_p$ forces us to equip input and output spaces with more general distances where the triangle inequality need not hold.

A novelty of our approach is that, to support the handling of such products, we generalize Fuzz environments to *bunches*, where each L^p distance comes with its own context former. Thus, we call our type system Bunched Fuzz. This system, inspired by languages derived from the logic of Bunched Implications (BI) [22] (e.g. [21]), highlights differences between the original Fuzz design and linear logic—for example, products distribute over sums in Fuzz and BI, but not in linear logic. While similar indexed products and function spaces have also appeared in the literature, particularly in works on categorical grammars [19], here they are employed to reason about vector distances and function sensitivity.

While designing Bunched Fuzz, one of our goals was to use sensitivity to reason about randomized algorithms. In the original Fuzz, probability distributions are equipped with the *max divergence* distance, which can be used to state differential privacy as a sensitivity property [23]. Subsequent work has shown how Fuzz can also accommodate other distances over probability distributions [3]. However, such additions required variants of *graded monads*, which express the distance between distributions using indices (i.e. grades) on the monadic type of distributions over their *results*, as opposed to sensitivity indices on their *inputs*, as it was done in the original Fuzz. In particular, this makes it more difficult to reason about distances separately with respect to each input. Thanks to bunches, however, we can incorporate these composition principles more naturally. For example, Bunched Fuzz can reason about the Hellinger distance on distributions without the need for output grading, as was done in prior systems [3].

We will also see that, by allowing arbitrary L^p norms, we can generalize prior case studies that were verified in Fuzz and obtain more general methods for reasoning about differential privacy (Section 5). Consider the L^p mechanism [1,17], which adds noise to the result of a query whose sensitivity is measured in the L^p norm. Since Fuzz does not have the means to analyze such a sensitivity measure, it cannot implement the L^p mechanism; Bunched Fuzz, however, can analyze such a measure, and thus allows for a simple implementation in terms of the exponential mechanism. Such a mechanism, in turn, can be used to implement a variant of a gradient descent algorithm that works under the L^p norm, generalizing an earlier version that was biased towards the L^1 norm [25]. Summarizing, our contributions are:

- We introduce Bunched Fuzz, an extension of Fuzz with types for general L^p distances: we add type constructors of the form $\otimes_p$ (for $1 \leq p \leq \infty$) for pairs under the L^p distance along with constructors of the form $\multimap_p$ for their corresponding function spaces. To support the handling of such types, we generalize Fuzz typing contexts to *bunches of variable assignments*.
- We give a denotational semantics for Bunched Fuzz by interpreting programs as non-expansive functions over spaces built on L^p distances.
- We show that Bunched Fuzz can support types for probability distributions for which the sampling primitive, which enables the composition of probabilistic programs, is compatible with L^p distances.
- We show a range of examples of programs that can be written in Bunched Fuzz. Notably, we show that Bunched Fuzz can support reasoning about the Hellinger distance without the need for grading, and we show generalizations of several examples from the differential privacy literature.

Check the full version of this paper for more technical details [26].

2 Background

2.1 Metrics and Sensitivity

To discuss sensitivity, we first need a notion of distance. We call *extended pseudosemimetric space* a pair $X = (|X|, d_X)$ consisting of a carrier set $|X|$ and an *extended pseudosemimetric* $d_X : |X|^2 \to \mathbb{R}^{\geq 0}_{\infty}$, which is a function satisfying, for all $x, y \in |X|$, $d_X(x, x) = 0$ and $d_X(x, y) = d_X(y, x)$. This relaxes the standard notion of metric space in a few respects. First, the distance between two points can be infinite, hence the *extended*. Second, different points can be at distance zero, hence the *pseudo*. Finally, we do not require the *triangular inequality*:

$$d_X(x, y) \leq d_X(x, z) + d_X(z, y), \tag{1}$$

hence the *semi*. We focus on extended pseudosemimetrics because they support constructions that true metrics do not. In particular, they make it possible to scale the distance of a space by ∞ and enable more general function spaces. However, to simplify the terminology, we will drop the "extended pseudosemi" prefix in the rest of the paper, and speak solely of metric spaces. In some occasions, we might speak of a *proper metric space*, by which we mean a space where the triangle inequality *does* hold (but not necessarily the other two requirements that are missing compared to the traditional definition of metric space).

Given a function $f : X \to Y$ on metric spaces, we say that it is s-sensitive, for s in $\mathbb{R}^{\geq 0}_{\infty}$, if we have, for all $x_1, x_2 \in X$, $d_Y(f(x_1), f(x_2)) \leq s \cdot d_X(x_1, x_2)$, where we extend addition and multiplication to $\mathbb{R}^{\geq 0}_{\infty}$ by setting $\infty \cdot s = s \cdot \infty = \infty$. We also say that f is *s-Lipschitz continuous*, though the traditional definition of Lipschitz continuity assumes $s \neq \infty$. If a function is s-sensitive, then it is also s'-sensitive for every $s' \geq s$. Every function of type $X \to Y$ is ∞-sensitive. If a function is 1-sensitive, we also say that f is non-expansive. We use $X \multimap Y$ to denote the set of such non-expansive functions. The identity function is always non-expansive, and non-expansive functions are closed under composition. Thus, metric spaces and non-expansive functions form a category, denoted Met.

2.2 Distances for Differential Privacy

Among many applications, sensitivity is a useful notion because it provides a convenient language for analyzing the privacy guarantees of algorithms—specifically, in the framework of differential privacy [11]. Differential privacy is a technique for protecting the privacy of individuals in a database by blurring the results of a query to the database with random noise. The noise is calibrated so that each individual has a small influence on the probability of observing each outcome (while ideally guaranteeing that the result of the query is still useful).

Formally, suppose that we have some set of databases db equipped with a metric. This metric roughly measures how many rows differ between two databases, though the exact definition can vary. Let $f : \mathsf{db} \to DX$ be a randomized database query, which maps a database to a discrete probability distribution over the set of outcomes X. We say that f is *ϵ-differentially private* if it is an ϵ-sensitive function from db to DX, where the set of distributions DX is equipped with the following distance, sometimes known as the *max divergence*:

$$\mathsf{MD}_X(\mu_1, \mu_2) = \sum_{x \in X} \ln \left| \frac{\mu_1(x)}{\mu_2(x)} \right|. \tag{2}$$

(Here, we stipulate that $\ln |0/0| = 0$ and $\ln |p/0| = \ln |0/p| = \infty$ for $p \neq 0$.)

To understand this definition, suppose that D_1 and D_2 are two databases at distance 1—for instance, because they differ with respect to the data of a single individual. If f is ϵ-differentially private, the above definition implies that $f(D_1)$ and $f(D_2)$ are at most ϵ apart. When ϵ is large, the probabilities of each outcome in the result distributions can vary widely. This means that, by simply observing one output of f, we might be able to guess with good confidence which of the databases D_1 or D_2 was used to produce that output. Conversely, if ϵ is small, it is hard to tell which database was used because the output probabilities will be close. For this reason, it is common to view ϵ as a *privacy loss*—the larger it is, the more privacy we are giving up to reveal the output of f.

Besides providing a strong privacy guarantee, this formulation of closeness for distributions provides two important properties. First, we can *compose* differentially private algorithms without ruining their privacy guarantee. Note that DX forms a monad, where the return and bind operations are given as follows:

$$\eta(x) = y \mapsto \begin{cases} 1 & \text{if } x = y \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$$f^\dagger(\mu) = y \mapsto \sum_{x \in X} \mu(x) \cdot f(x)(y). \tag{4}$$

Intuitively, the return operation produces a deterministic distribution, whereas bind samples an element x from μ and computes $f(x)$. When composing differentially private algorithms, their privacy loss can be soundly added together:

Theorem 1. *Suppose that $f : \mathsf{db} \to DX$ is ϵ_1-differentially private and that $g : \mathsf{db} \to X \to DY$ is such that the mapping $\delta \to g(\delta)(x)$ is ϵ_2-differentially private for every x. Then the composite $h : \mathsf{db} \to DY$ defined as $h(\delta) = g(\delta)^\dagger(f(\delta))$ is $(\epsilon_1 + \epsilon_2)$-differentially private.*

The other reason why the privacy metric is useful is that it supports many building blocks for differential privacy. Of particular interest is the *Laplace mechanism*, which blurs a numeric result with noise drawn from the two-sided Laplace distribution. If $x \in \mathbb{R}$, let $\mathcal{L}(x)$ be the distribution with density[4] $y \mapsto \frac{1}{2}e^{-|x-y|}$.

Theorem 2. *The mechanism $\mathcal{L}$ is a non-expansive function of type $\mathbb{R} \to D\mathbb{R}$.*[5]

Thus, to define an ϵ-differentially private numeric query on a database, it suffices to define an ϵ-sensitive, deterministic numeric query, and then blur its result with Laplace noise. Differential privacy follows from the composition principles for sensitivity. This reasoning is justified by the fact that the Laplace mechanism adds noise proportional to the sensitivity of the numeric query in L^1 distance.

2.3 Sensitivity as a Resource

Because differential privacy is a sensitivity property, techniques for analyzing the sensitivity of programs can also be used to analyze their privacy guarantees. One particularly successful approach in this space is rooted in type systems inspired by linear logic, as pioneered by Reed and Pierce in the Fuzz programming language [16,23]. At its core, Fuzz is just a type system for tracking sensitivity.

[4] We use here a Laplace distribution with scale 1.

[5] The definitions do not quite match up our setting, since $\mathcal{L}$ is a continuous, and not discrete distribution. The result can be put on firm footing by working with a discretized version of the Laplace distribution [12].

Typing judgments are similar to common functional programming languages, but variable declarations are of the form $x_i :_{r_i} \tau_i$: $x_1 :_{r_1} \tau_1, \ldots, x_n :_{r_n} \tau_n \vdash e : \sigma$. The annotations $r_i \in \mathbb{R}^{\geq 0}_{\infty}$ are *sensitivity indices*, whose purpose is to track the effect that changes to the program input can have on its output: if we have two substitutions γ and γ' for the variables x_i, then the *metric preservation* property of the Fuzz type system guarantees that

$$d(e[\gamma/\vec{x}], e[\gamma'/\vec{x}]) \leq \sum_i r_i \cdot d(\gamma(x_i), \gamma'(x_i)), \tag{5}$$

where the metrics d are computed based on the type of each expression and value. This means that we can bound the distance on the results of the two runs of e by adding up the distances of the inputs scaled by their corresponding sensitivities. When this bound is finite, the definition of the metrics guarantees that the two runs have the same termination behavior. When $r_i = \infty$, the above inequality provides no guarantees if the value of x_i varies.

Fuzz includes data types commonly found in functional programming languages, such as numbers, products, tagged unions, recursive types and functions. The typing rules of the language explain how the sensitivities of each variable must be updated to compute each operation. The simplest typing rule says that, in order to use a variable, its declared sensitivity must be greater than 1:

$$\frac{r \geq 1}{\Gamma, x :_r \tau, \Delta \vdash x : \tau}$$

As a more interesting example, to construct a pair (e_1, e_2), the following rule says that we need to add the sensitivities of the corresponding contexts:

$$\frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2 \vdash e_2 : \tau_2}{\Gamma_1 + \Gamma_2 \vdash (e_1, e_2) : \tau_1 \otimes \tau_2}.$$

This behavior is a result of the distance of the tensor type $\otimes$: the distance between two pairs in $\tau_1 \otimes \tau_2$ is the result of adding the distances between the first and second components; therefore, the sensitivity of each variable for the entire expression is the sum of the sensitivities for each component. In this sense, sensitivities in Fuzz behave like a resource that must be distributed across all variable uses in a program. For the sake of analogy, we might compare this treatment to how fractional permissions work in separation logic: the predicate $l \mapsto_q x$ indicates that we own a fraction $q \in [0, 1]$ of a resource stating that l points to x. If $q = q_1 + q_2$, we can split this predicate as $l \mapsto_{q_1} x * l \mapsto_{q_2} x$, allowing us to distribute this resource between different threads.

The distance on $\otimes$ corresponds to the sum in the upper bound in the statement of metric preservation (Equation (5)). This distance is useful because it is the one that yields good composition principles for differential privacy. This can be seen in the typing rule for sampling from a probabilistic distribution:

$$\frac{\Gamma \vdash e_1 : \bigcirc\tau \qquad \Delta, x :_r \tau \vdash e_2 : \bigcirc\sigma}{\Gamma + \Delta \vdash \mathbf{mlet}\ x = e_1\ \mathbf{in}\ e_2 : \bigcirc\sigma}$$

Here, $\bigcirc\tau$ denotes the type of probability distributions over values of type τ. This operation samples a value x from the distribution e_1 and uses this value to compute the distribution e_2. We can justify the soundness of this rule by reducing it to Theorem 1: the addition on contexts corresponds to the fact that the privacy loss of a program degrades linearly under composition.

Besides the tensor product $\otimes$, Fuzz also features a *with product* &, where the distances between components are combined by taking their maximum. This leads to a different typing rule for & pairs, which does not add up the sensitivities:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \mathbin{\&} \tau_2}$$

If we compare these rules for pairs, we see a clear analogy with linear logic: $\otimes$ requires us to combine contexts, whereas & allows us to share them. Fuzz's elimination rules for products continue to borrow from linear logic: deconstructing a tensor gives both elements but deconstructing a with product returns only one.

$$\frac{\Gamma \vdash e : \tau_1 \otimes \tau_2 \qquad \Delta, x :_r \tau_1, y :_r \tau_2 \vdash e' : \tau'}{\Delta + r\Gamma \vdash \textbf{let } (x, y) = e \textbf{ in } e' : \tau'} \qquad\qquad \frac{\Gamma \vdash e : \tau_1 \mathbin{\&} \tau_2}{\Gamma \vdash \pi_i\ e : \tau_i}$$

This partly explains why the connectives' distances involve addition and maximum. When using a tensor product, both elements can affect how much the output can vary, so both elements must be considered. (Note that Fuzz is an affine type system: we are free to ignore one of the product's components, and thus we can write projection functions out of a tensor product.) When projecting out of a with product, only one of the elements will affect the program's output, so we only need to consider the component that yields the maximum distance.

Fuzz uses the $!_s$ type for managing sensitivities. Intuitively, $!_s\tau$ behaves like τ, but with the distances scaled by s; when $s = \infty$, this means that different points are infinitely apart. The introduction rule scales the sensitivities of variables in the environment. This can be used in conjunction with the elimination rule to propagate the sensitivity out of the type and into the environment.

$$\frac{\Gamma \vdash e : \tau}{s\Gamma \vdash\ !e : !_s\tau} \qquad\qquad \frac{\Gamma \vdash e : !_s\tau \qquad \Delta, x :_{rs} \tau \vdash e' : \tau'}{\Delta + r\Gamma \vdash \textbf{let } !x = e \textbf{ in } e' : \tau'}$$

Finally, the rules for the linear implication $\multimap$ are similar to the ones from linear logic, but adjusted to account for sensitivities.

$$\frac{\Gamma, x :_1 \tau \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \tau \multimap \sigma} \qquad\qquad \frac{\Gamma \vdash e : \tau \multimap \sigma \qquad \Delta \vdash e' : \tau}{\Gamma + \Delta \vdash e\ e' : \sigma}$$

To introduce the linear implication $\multimap$, the bound variable needs to have sensitivity 1. When eliminating $\multimap$, the environments need to be added. In categorical language, addition, which is also present in the metric for $\otimes$, is connected to the fact that there is an adjunction between the functors $X \otimes (-)$ and $X \multimap (-)$.

2.4 L^p distances

The L^1 and L^∞ distances are instances of a more general family of L^p distances (for $p \in \mathbb{R}^{\geq 1}_{\infty}$).[6] Given a sequence of distances $\vec{x} = (x_1, \ldots, x_n) \in (\mathbb{R}^{\geq 0}_{\infty})^n$, we first define the L^p *pseudonorm*[7] as follows: $||\vec{x}||_p = (\Sigma_{i=1}^n x_i^p)^{1/p}$. This definition makes sense whenever the distances x_i and p are finite. When $p = \infty$, we define the right-hand side as the limit $\max_{i=1}^n x_i$. When $x_i = \infty$ for some i, we define the right-hand side as ∞. We have the following classical properties:

Proposition 1 (Hölder inequality). *For all $p, q \geq 1$ such that $\frac{1}{p} + \frac{1}{q} = 1$, and for all $\vec{x}$, $\vec{y} \in (\mathbb{R}^{\geq 0}_{\infty})^n$, we have: $\Sigma_{i=1}^n x_i y_i \leq ||\vec{x}||_p ||\vec{y}||_q$.*
For $p = 2$, $q = 2$, this is the Cauchy-Schwarz inequality: $\Sigma_{i=1}^n x_i y_i \leq ||\vec{x}||_2 ||\vec{y}||_2$.

Proposition 2. *For $1 \leq p \leq q$ we have, for $\vec{x} \in (\mathbb{R}^{\geq 0}_{\infty})^n$:*

$$||\vec{x}||_q \leq ||\vec{x}||_p \tag{6}$$

$$||\vec{x}||_p \leq n^{\frac{1}{p} - \frac{1}{q}} ||\vec{x}||_q \tag{7}$$

$$||\vec{x}||_2 \leq ||\vec{x}||_1 \leq \sqrt{n}\, ||\vec{x}||_2 \tag{8}$$

The L^p pseudonorms yield distances on tuples. More precisely, suppose that $(X_i)_{1 \leq i \leq n}$ are metric spaces. The following defines a metric on $X = X_1 \times \cdots \times X_n$:

$$d_p(\vec{x}, \vec{x}') = ||(d_{X_1}(x_1, x_1'), \ldots, d_{X_n}(x_n, x_n'))||_p$$

Proposition 3. *For $1 \leq p \leq q$ we have, for $\vec{x}, \vec{x'} \in X_1 \times \cdots \times X_n$:*

$$d_q(\vec{x}, \vec{x'}) \leq d_p(\vec{x}, \vec{x'}) \leq n^{\frac{1}{p} - \frac{1}{q}} d_q(\vec{x}, \vec{x'}) \tag{9}$$

$$d_2(\vec{x}, \vec{x'}) \leq d_1(\vec{x}, \vec{x'}) \leq \sqrt{n}\, d_2(\vec{x}, \vec{x'}) \tag{10}$$

3 Bunched Fuzz: Programming with L^p Distances

As we discussed earlier, the L^1 distance is not the only distance on products with useful applications. In the context of differential privacy, for example, the L^2 distance is used to measure the sensitivity of queries when employing the Gaussian mechanism, a method for private data release that sanitizes data by adding Gaussian noise instead of Laplacian noise.[8]

It is possible to extend a Fuzz-like analysis with L^2 distances by adding primitive types and combinators for vectors. This was done, for instance, in

[6] The L^p distances can be defined with $p \geq 0$ but for simplicity of our treatment we will only consider $p \geq 1$.

[7] "pseudo-" because it can be infinite.

[8] Technically, the Gaussian mechanism is used to achieve a relaxation of differential privacy known as *approximate*, or (ϵ, δ)-differential privacy. Though this notion cannot be analyzed directly by classical verification techniques for differential privacy, it can be handled by recent extensions of Fuzz [3,20].

the Duet language [20], which provides the Gaussian mechanism as one of the primitives for differential privacy. Such an extension can help verify a wide class of algorithms that manipulate vectors in a homogeneous fashion, but it makes it awkward to express programs that require finer grained access to vectors.

To illustrate this point, suppose that we have a non-expansive function $f : \mathbb{R}^2 \to \mathbb{R}$, where the domain carries the L^2 metric. Consider the mapping

$$g(x, y) = f(2x, y) + f(2y, x).$$

How would we analyze the sensitivity of g? We cannot translate such a program directly into a system like Duet, since it does not allow us to manipulate L^2 vectors at the level of individual components. However, we could rewrite the definition of g to use matrix operations, which could be easily incorporated in a variant of Duet. Specifically, consider the following definition:

$$g(\vec{x}) = f\left(\begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} \vec{x}\right) + f\left(\begin{bmatrix} 0 & 2 \\ 1 & 0 \end{bmatrix} \vec{x}\right).$$

The L^2 sensitivity of a linear transformation $\vec{x} \mapsto M\vec{x}$ can be easily computed if we know the coefficients of the matrix M. Note that

$$\begin{aligned} d(M\vec{x}, M\vec{y}) = ||M\vec{x} - M\vec{y}||_2 = ||M(\vec{x} - \vec{y})||_2 &= \frac{||M(\vec{x} - \vec{y})||_2}{||\vec{x} - \vec{y}||_2} ||\vec{x} - \vec{y}||_2 \\ &\leq \left(\sup_{\vec{z}} \frac{||M\vec{z}||_2}{||\vec{z}||_2}\right) d(\vec{x}, \vec{y}). \end{aligned}$$

The quantity $\sup_{\vec{z}} ||M\vec{z}||_2/||\vec{z}||_2$, known as the *operator norm* of M, gives the precise sensitivity of the above operation, and can be computed by standard algorithms from linear algebra. In the case of g, both matrices have a norm of 2. This means that we can analyze the sensitivity of g compositionally, as in Fuzz: addition is 1-sensitive in each variable, so we just have to sum the sensitivities of $\vec{x}$ in each argument, yielding a combined sensitivity of 4. Unfortunately, this method of combining the sensitivities of each argument is too coarse when reasoning with L^p distances, which leads to an imprecise analysis. To obtain a better bound, we can reason informally as follows. First, take $M = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}^T$. We can compute the operator norm of M directly:

$$||M|| = \sup_{x,y} \frac{\sqrt{2^2x^2 + y^2 + 2^2y^2 + x^2}}{\sqrt{x^2 + y^2}} = \sup_{x,y} \frac{\sqrt{5(x^2 + y^2)}}{\sqrt{x^2 + y^2}} = \sqrt{5},$$

which implies that M is a $\sqrt{5}$-sensitive function of type $\mathbb{R}^2 \to \mathbb{R}^4 \cong \mathbb{R}^2 \times \mathbb{R}^2$. Moreover, thanks to Proposition 3, we can view addition $(+)$ as a $\sqrt{2}$-sensitive operator of type $\mathbb{R}^2 \to \mathbb{R}$, since

$$d_{\mathbb{R}}(x_1 + x_2, y_1 + y_2) \leq d_{\mathbb{R}}(x_1 - y_1) + d_{\mathbb{R}}(x_2 - y_2) = d_1(\vec{x}, \vec{y}) \leq \sqrt{2} d_2(\vec{x}, \vec{y}).$$

$$
\begin{aligned}
\tau, \sigma, \rho &::= 1 \mid \mathbb{R} \mid !_s \tau \mid \bigcirc_P \tau \mid \bigcirc_H \tau \mid \tau \multimap_p \sigma \mid \tau \otimes_p \sigma \mid \tau \oplus \sigma \qquad (p \in \mathbb{R}^{\geq 1}_{\infty}, s \in \mathbb{R}^{\geq 0}_{\infty}) \\
e &::= x \mid r \in \mathbb{R} \mid () \mid \lambda x.e \mid e\ e \mid (e, e) \mid \textbf{let}\ (x, y) = e\ \textbf{in}\ e \\
&\mid \textbf{inj}_i e \mid (\textbf{case}\ e\ \textbf{of}\ x.\ e \mid y.\ e) \mid !e \mid \textbf{let}\ !x = e\ \textbf{in}\ e \\
&\mid \textbf{mlet}\ x = e\ \textbf{in}\ e \mid \textbf{return}\ e \mid \cdots
\end{aligned}
$$

Fig. 1. Types and terms in Bunched Fuzz

Thus, by rewriting the definition of g as $(+) \circ (f \times f) \circ M$, where $f \times f : \mathbb{R}^4 \cong \mathbb{R}^2 \times \mathbb{R}^2 \to \mathbb{R} \times \mathbb{R}$ denotes the application of f in parallel, we can compute the sensitivity of g by multiplying the sensitivity of each stage, as $\sqrt{2} \times 1 \times \sqrt{5} = \sqrt{10} \approx 3.16$, which is strictly better than the previous bound.

Naturally, we could further extend Fuzz or Duet with primitives for internalizing this reasoning, but it would be preferable to use the original definition of g and automate the low-level reasoning about distances. In this section, we demonstrate how this can be done via Bunched Fuzz, a language that refines Fuzz by incorporating more general distances in its typing environments. Rather assuming that input distances are always combined by addition, or the L^1 distance, Bunched Fuzz allows them to be combined with arbitrary L^p distances. This refinement allows us to analyze different components of a vector as individual variables, but also to split the sensitivity of these variables while accounting for their corresponding vector distances. In the remaining of this section, we present the syntax and type system of Bunched Fuzz, highlighting the main differences with respect to the original Fuzz design. Later, in Section 4, we will give a semantics to this language in terms of metric spaces, following prior work [3].

Types and Terms Figure 1 presents the grammar of types and the main term formers of Bunched Fuzz. They are similar to their Fuzz counterparts; in particular, there are types for real numbers, products, sums, functions, and a unit type. The main novelty is in the product type $\tau \otimes_p \sigma$, which combines the metrics of each component using the L^p distance (cf. Section 2.4). The types $\tau \otimes_1 \sigma$ and $\tau \otimes_\infty \sigma$ subsume the types $\tau \otimes \sigma$ and $\tau \,\&\, \sigma$ in the original Fuzz language. Note that there is no term constructor or destructor for the Fuzz type &, since it is subsumed by $\otimes_\infty$. The type $\tau \multimap_p \sigma$ represents non-expansive functions endowed with a metric that is compatible with the L^p metric, in that currying works (cf. Section 5). We will sometimes write $\otimes$ for $\otimes_1$ and $\multimap$ for $\multimap_1$.

Another novelty with respect to Fuzz is that there are two constructors for probability distributions, $\bigcirc_P$ and $\bigcirc_H$. The first one carries the original Fuzz privacy metric, while the second one carries the Hellinger distance. As we will see shortly, the composition principle for the Hellinger distance uses a contraction operator for the L^2 distance, which was not available in the original Fuzz design. Both distribution types feature term constructors **mlet** and **return** for sampling from a distribution and for injecting values into distributions. To simplify the notation, we do not use separate versions of these term formers for each type.

Bunches Before describing its type system, we need to talk about how typing environments are handled in Bunched Fuzz. In the spirit of bunched logics, environments are bunches defined with the following grammar:

$$\Gamma, \Delta ::= \cdot \mid [x : \tau]_s \mid \Gamma \,,_p \Delta$$

The empty environment is denoted as $\cdot$. The form $[x : \tau]_s$ states that the variable x has type τ and sensitivity s. The form $\Gamma \,,_p \Delta$ denotes the concatenation of Γ and Δ, which is only defined when the two bind disjoint sets of variables. As we will see in Section 4, bunches will be interpreted as metric spaces, and the p index denote which L^p metric we will use to combine the metrics of Γ and Δ.

The type system features several operations and relations on bunches, which are summarized in Figure 2. We write $\Gamma \leftrightsquigarrow \Gamma'$ to indicate that we can obtain Γ' by rearranging commas up to associativity and commutativity, and by treating the empty environment as an identity element; Figure 2 has a precise definition. Observe that associativity only holds for equal values of p. This operation will be used to state a permutation rule for the type system of Bunched Fuzz.

Like in Fuzz, environments have a scaling operation $s\Gamma$ which scales all sensitivities in the bunch by s. For example,

$$s([x : \tau]_{r_1} \,,_p [y : \sigma]_{r_2}) = ([x : \tau]_{s \cdot r_1} \,,_p [y : \sigma]_{s \cdot r_2}).$$

The exact definition of scaling in such graded languages is subtle, since minor variations can quickly lead to unsoundness. The definition we are using ($\infty \cdot 0 = 0 \cdot \infty = \infty$), which goes back to prior work [3], is sound, but imprecise, since it leads to too many variables being marked as ∞-sensitive. It would also be possible to have a more precise variant that uses a non-commutative definition of multiplication on distances [4], but we keep the current formulation for simplicity. (For a more thorough discussion on these choices and their tradeoffs, see the “Zero and Infinity” example in Appendix B of the full version [26] of this paper.)

In the original Fuzz type system, rules with several premises usually have their environments combined by adding sensitivities pointwise, which corresponds to a use of the L^1 metric. In Bunched Fuzz, we have instead a family of contraction operations $\mathit{Contr}(p, \Gamma, \Delta)$ for combining environments, one for each L^p metric. Contraction only makes sense if Γ and Δ differ only in sensitivities and variable names, but have the same structure otherwise. We write this relation as $\Gamma \approx \Delta$. When contracting two leaves, sensitivities are combined using the L^p norm, while keeping variable names from the left bunch.

Unlike Fuzz, where contraction is implicit in rules with multiple premises, Bunched Fuzz has a separate, explicit contraction typing rule. The rule will be stated using the *vars* function, which lists all variables in a bunch.

Type System Our type system is similar to the one of Fuzz, but adapted to use bunched environments. The typing rules are displayed on Figure 3. For example, in the $\otimes$I rule, notice that the p on the tensor type is carried over to the bunch in the resulting environment. Similarly, in the $\multimap$I rule, the value of p that annotates the bunch in the premise is carried over to the $\multimap_p$ in the conclusion.

$$
\begin{array}{rcl}
vars(\cdot) = [] & \cdot \approx \cdot & \\
vars([x:\tau]_s) = [x] & [x:\tau]_s \approx [y:\sigma]_r & \text{if } \tau = \sigma \\
vars((\Gamma_1,_p \Gamma_2)) = vars(\Gamma_1) \mathbin{+\!\!+} vars(\Gamma_2) & \Gamma_1 ,_p \Gamma_2 \approx \Delta_1,_q \Delta_2 & \text{if } p = q \wedge \Gamma_i \approx \Delta_i
\end{array}
$$

$$
\begin{array}{rlc}
\Gamma \leftrightsquigarrow \Delta & \text{if } \Gamma = \Delta & \\
\Gamma \leftrightsquigarrow \cdot,_p \Delta & \text{if } \Gamma \leftrightsquigarrow \Delta & \\
\Gamma \leftrightsquigarrow \Delta,_p \cdot & \text{if } \Gamma \leftrightsquigarrow \Delta & s\,\cdot = \cdot \\
\Gamma_1,_p \Gamma_2 \leftrightsquigarrow \Delta_1,_p \Delta_2 & \text{if } \Gamma_i \leftrightsquigarrow \Delta_i & s\,[\tau]_r = [\tau]_{s\cdot r} \\
\Gamma_1,_p \Gamma_2 \leftrightsquigarrow \Delta_2,_p \Delta_1 & \text{if } \Gamma_i \leftrightsquigarrow \Delta_i & s\,(\Gamma ,_p \Delta) = s\Gamma ,_p s\Delta \\
\Gamma_1,_p (\Gamma_2,_p \Gamma_3) \leftrightsquigarrow (\Delta_1,_p \Delta_2),_p \Delta_3 & \text{if } \Gamma_i \leftrightsquigarrow \Delta_i & \\
\Gamma_2 \leftrightsquigarrow \Gamma_1 & \text{if } \Gamma_1 \leftrightsquigarrow \Gamma_2 &
\end{array}
$$

$$
\begin{aligned}
c(p,q) &= \begin{cases} 1 & \text{if } p = \infty \\ 2^{\left|\frac{1}{q} - \frac{1}{p}\right|} & \text{otherwise} \end{cases} \\
Contr(p, \cdot, \cdot) &= \cdot \\
Contr(p, [x:\tau]_s, [y:\tau]_r) &= [x:\tau]_{\sqrt[p]{s^p + r^p}} \\
Contr(p, (\Gamma_1,_q \Gamma_2), (\Delta_1,_q \Delta_2)) &= c(p,q)(Contr(p, \Gamma_1, \Delta_1),_q Contr(p, \Gamma_2, \Delta_2)).
\end{aligned}
$$

Fig. 2. Bunch Operations

Like in Fuzz, the !E rule propagates the scaling factor, but using the bunch structure. Rather than adding the two environments, we splice one into the other: the notation $\Gamma(\Delta)$ denotes a compound bunch where we plug in the bunch Δ into another bunch $\Gamma(\star)$ that has a single, distinguished hole $\star$. As we mentioned earlier, Bunched Fuzz has an explicit typing rule for contraction, whereas contraction in Fuzz is implicit in rules with multiple premises. Note also that we have unrestricted weakening. Finally, we have the rules for typing the return and bind primitives of the probabilistic types $\bigcirc_H$ and $\bigcirc_P$. Those for $\bigcirc_P$ are adapted from Fuzz, by using contraction instead of adding up the environments. The ones for $\bigcirc_H$ are similar, but use L^2 contraction instead, since that is the metric that enables composition for the Hellinger distance.

Let us now explain in which sense $\otimes_\infty$ corresponds to the & connective of Fuzz. We will need the following lemma:

Lemma 1 (Renaming). *Assume that there is a type derivation of $\Gamma \vdash e : \tau$ and that $\Gamma \approx \Gamma'$. Then there exists a derivation of $\Gamma' \vdash e[vars(\Gamma')/vars(\Gamma)] : \tau$.*

Now, the & connective in Fuzz supports two operations, projections and pairing. The connective $\otimes_\infty$ of Bunched Fuzz also supports these operations, but as derived forms. First, projections can be encoded by defining $\pi_i(e)$ for $i = 1, 2$ as **let** $(x_1, x_2) = e$ **in** x_i. Second, for pairing assume we have two derivations of $\Gamma \vdash e_i : \sigma_i$ for $i = 1, 2$, and let Γ' be an environment obtained from Γ by

$$\frac{s \geq 1}{[x : \tau]_s \vdash x : \tau}\ \textsc{Axiom} \qquad \frac{}{\cdot \vdash r : \mathbb{R}}\ \mathbb{R}\mathrm{I} \qquad \frac{}{\cdot \vdash () : 1}\ 1\mathrm{I}$$

$$\frac{\Gamma ,_p [x : \tau]_1 \vdash e : \sigma}{\Gamma \vdash \lambda x.e : \tau \multimap_p \sigma}\ \multimap\mathrm{I} \qquad \frac{\Gamma \vdash f : \tau \multimap_p \sigma \qquad \Delta \vdash e : \tau}{\Gamma ,_p \Delta \vdash f\ e : \sigma}\ \multimap\mathrm{E}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Delta \vdash e_2 : \sigma}{\Gamma ,_p \Delta \vdash (e_1, e_2) : \tau \otimes_p \sigma}\ \otimes\mathrm{I} \qquad \frac{\Delta \vdash e_1 : \tau \otimes_p \sigma \qquad \Gamma([x : \tau]_s ,_p [y : \sigma]_s) \vdash e_2 : \rho}{\Gamma(s\Delta) \vdash \mathbf{let}\ (x, y) = e_1\ \mathbf{in}\ e_2 : \rho}\ \otimes\mathrm{E}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathbf{inj}_1 e : \tau \oplus \sigma}\ \oplus_1\mathrm{I} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathbf{inj}_2 e : \tau \oplus \sigma}\ \oplus_2\mathrm{I}$$

$$\frac{\Gamma \vdash e_1 : \tau \oplus \sigma \qquad \Delta([x : \tau]_s) \vdash e_2 : \rho \qquad \Delta([y : \sigma]_s) \vdash e_3 : \rho}{\Delta(s\Gamma) \vdash \mathbf{case}\ e_1\ \mathbf{of}\ x.\ e_2 \mid y.\ e_3 : \rho}\ \oplus\mathrm{E}$$

$$\frac{\Gamma \vdash e : \tau}{s\Gamma \vdash !e : !_s\tau}\ !\mathrm{I} \qquad \frac{\Gamma \vdash e_1 : !_r\tau \qquad \Delta([x : \tau]_{rs}) \vdash e_2 : \sigma}{\Delta(s\Gamma) \vdash \mathbf{let}\ !x = e_1\ \mathbf{in}\ e_2 : \sigma}\ !\mathrm{E}$$

$$\frac{\Gamma(\Delta ,_p \Delta') \vdash e : \tau \qquad \Delta \approx \Delta'}{\Gamma(Contr(p, \Delta, \Delta')) \vdash e[vars(\Delta')/vars(\Delta)] : \tau}\ \textsc{Contr} \qquad \frac{\Gamma(\cdot) \vdash e : \tau}{\Gamma(\Delta) \vdash e : \tau}\ \textsc{Weak}$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \leftrightsquigarrow \Gamma'}{\Gamma' \vdash e : \tau}\ \textsc{Exch}$$

$$\frac{\begin{array}{c} \Gamma \approx \Delta \\ \Gamma \vdash e_1 : \bigcirc_P\tau \qquad \Delta ,_p [x : \tau]_s \vdash e_2 : \bigcirc_P\sigma \end{array}}{Contr(1, \Gamma, \Delta) \vdash \mathbf{mlet}\ x = e_1\ \mathbf{in}\ e_2 : \bigcirc_P\sigma}\ \textsc{Bind-P} \frac{\Gamma \vdash e : \tau}{\infty\Gamma \vdash \mathbf{return}\ e : \bigcirc_P\tau}\ \textsc{Return-P}$$

$$\frac{\begin{array}{c} \Gamma \approx \Delta \\ \Gamma \vdash e_1 : \bigcirc_H\tau \qquad \Delta ,_p [x : \tau]_s \vdash e_2 : \bigcirc_H\sigma \end{array}}{Contr(2, \Gamma, \Delta) \vdash \mathbf{mlet}\ x = e_1\ \mathbf{in}\ e_2 : \bigcirc_H\sigma}\ \textsc{Bind-H} \frac{\Gamma \vdash e : \tau}{\infty\Gamma \vdash \mathbf{return}\ e : \bigcirc_H\tau}\ \textsc{Return-H}$$

Fig. 3. Bunched Fuzz typing rules

renaming all variables to fresh ones. Then we have $\mathit{\Gamma} \approx \mathit{\Gamma}'$ and thus

$$\dfrac{\dfrac{\mathit{\Gamma} \vdash e_1 : \sigma_1 \qquad \dfrac{\mathit{\Gamma} \vdash e_2 : \sigma_2 \qquad \mathit{\Gamma} \approx \mathit{\Gamma}'}{\mathit{\Gamma}' \vdash e_2[vars(\mathit{\Gamma}')/vars(\mathit{\Gamma})] : \sigma_2}\ \textsc{Lemma } 1}{\mathit{\Gamma} ,_\infty \mathit{\Gamma}' \vdash (e_1, e_2[vars(\mathit{\Gamma}')/vars(\mathit{\Gamma})]) : \sigma_1 \otimes_\infty \sigma_2}\ \otimes\text{I}}{Contr(\infty, \mathit{\Gamma}, \mathit{\Gamma}') \vdash (e_1, e_2) : \sigma_1 \otimes_\infty \sigma_2}\ \textsc{Contr}$$

Note that we have defined $\sqrt[\infty]{x^\infty + y^\infty} = \max(x, y)$ by taking the limit of $\sqrt[p]{x^p + y^p}$ when p goes to infinity, and thus we have $Contr(\infty, \mathit{\Gamma}, \mathit{\Gamma}') = \mathit{\Gamma}$. Therefore the pairing rule of & is derivable for $\otimes_\infty$.

4 Semantics

Having defined the syntax of Bunched Fuzz and its type system, we are ready to present its semantics. We opt for a denotational formulation, where types τ and bunches $\mathit{\Gamma}$ are interpreted as metric spaces $[\![\tau]\!]$ and $[\![\mathit{\Gamma}]\!]$, and a derivation π of $\mathit{\Gamma} \vdash e : \tau$ is interpreted as a non-expansive function $[\![\pi]\!] : [\![\mathit{\Gamma}]\!] \to [\![\tau]\!]$. For space reasons, we do not provide an operational semantics for the language, but we foresee no major difficulties in doing so, since the term language is mostly inherited from Fuzz, which does have a denotational semantics proved sound with respect to an operational semantics [4].

Types Each type τ is interpreted as a metric space $[\![\tau]\!]$ in a compositional fashion, by mapping each type constructor to the corresponding operation on metric spaces defined in Figure 4. We now explain these definitions.

The operations of the first four lines of Figure 4 come from prior work on Fuzz [4,3]. The definition of $\otimes_p$ uses as carrier set the cartesian product, just as $\otimes$ in previous works, but endows it with the L^p distance, defined in Section 2.4. In the particular case of $p = 1$, $\otimes_1$ is the same as $\otimes$.

As for $\multimap_p$, we want to define it in such a way that currying and uncurrying work with respect to $\otimes_p$, which will allow us to justify the introduction and elimination forms for that connective. For that we first choose as carrier set the set $A \multimap B$ of non-expansive functions from A to B. This set carries the metric

$$\begin{aligned} & d_{A \multimap_p B}(f, g) \\ & = \inf\{r \in \mathbb{R}^{\geq 0}_{\infty} \mid \forall x, y \in A, d_B(f(x), g(y)) \leq \sqrt[p]{r^p + d_A(x, y)^p}\} \end{aligned} \tag{11}$$

This metric is dictated by the type of the application operator in the L^p norm: $(A \multimap_p B) \otimes_p A \multimap B$. Intuitively, if f and g are at distance r, and we want application to be non-expansive, we need to satisfy $d_B(f(x), g(y)) \leq \sqrt[p]{r^p + d_A(x, y)^p}$ for every $x, y \in A$. The above definition says that we pick the distance to be the smallest possible r that makes this work. Note that this choice is forced upon us: in category-theoretic jargon, the operations of currying and uncurrying, which are intimately tied to the application operator, correspond to an adjunction between two functors, which implies that any other metric space that yields a similar adjunction with respect to $\otimes_p$ must be isomorphic to $\multimap_p$. In particular, this implies that its metric will be the same as the one of $\multimap_p$.

For $\bigcirc_P A$ and $\bigcirc_H A$ the carrier set is the set DA of discrete distributions over A. As to the metric on the carrier set, the interpretation of $\bigcirc_P$ uses the max divergence, used in the definition of differential privacy (see Sect. 2.2). The interpretation of $\bigcirc_H$ uses instead the Hellinger distance (see e.g. [3]):

$$\mathsf{HD}_A(\mu,\nu) \triangleq \sqrt{\frac{1}{2}\sum_{x\in A} |\sqrt{\mu(x)} - \sqrt{\nu(x)}|^2} \tag{12}$$

Space X	$\lvert X\rvert$	$d_X(x,y)$
1	$\{*\}$	0
$\mathbb{R}$	$\mathbb{R}$	$\lvert x-y\rvert$
$!_s A$	$\lvert A\rvert$	$\begin{cases} s\cdot d_A(x,y) \text{ if } s\neq\infty \\ \infty \text{ if } s=\infty, x\neq y\in A \\ 0 \text{ if } s=\infty, x=y\in A \end{cases}$
$A\oplus B$	$\lvert A\rvert + \lvert B\rvert$	$\begin{cases} d_A(x,y) \text{ if } x,y\in A \\ d_B(x,y) \text{ if } x,y\in B \\ \text{else } \infty \end{cases}$
$A\otimes_p B$	$\lvert A\rvert \times \lvert B\rvert$	$\sqrt[p]{d_A(\pi_1(x),\pi_1(y))^p + d_B(\pi_2(x),\pi_2(y))^p}$
$A \multimap_p B$	$A\multimap B$	cf. Equation (11)
$\bigcirc_P A$	DA	$\mathsf{MD}_A(x,y)$; cf. Equation (2)
$\bigcirc_H A$	DA	$\mathsf{HD}_A(x,y)$; cf. Equation (12)

Fig. 4. Operations on metric spaces for interpreting types

Bunches The interpretation of bunches is similar to that of types. Variables correspond to scaled metric spaces, whereas $,_p$ corresponds to $\otimes_p$:

$$[\![\cdot]\!] = 1 \qquad [\![[x:\tau]_s]\!] = !_s[\![\tau]\!] \qquad [\![\Gamma_1 ,_p \Gamma_2]\!] = [\![\Gamma_1]\!] \otimes_p [\![\Gamma_2]\!].$$

One complication compared to prior designs is the use of an explicit exchange rule, which is required to handle the richer structure of contexts. Semantically, each use of exchange induces an isomorphism of metric spaces:

Theorem 3. *Each derivation of $\Gamma \leftrightsquigarrow \Delta$ corresponds to an isomorphism of metric spaces $[\![\Gamma]\!] \cong [\![\Delta]\!]$.*

Before stating the interpretation of typing derivations, we give an overview of important properties of the above constructions that will help us prove the soundness of the interpretation.

Scaling Much like in prior work [4,3], we can check the following equations:

Proposition 4.

$$!_{s_1}!_{s_2}A = \,!_{s_1 \cdot s_2}A \qquad !_s(A \oplus B) = !_sA \oplus !_sB \qquad !_s(A \otimes_p B) = !_sA \otimes_p !_sB.$$

Moreover, an s-sensitive function from A to B is the same thing as a non-expansive function of type $!_sA \multimap B$.

Proposition 5. *For every bunch Γ, we have $[\![s\Gamma]\!] = !_s[\![\Gamma]\!]$.*

Tensors The properties on L^p distances allow us to relate product types with different values of p.

Proposition 6. *[Subtyping of tensors]*

1. *Let A, B be two metric spaces and $p, q \in \mathbb{R}^{\geq 1}_{\infty}$ with $p \leq q$. Then the identity map on pairs belongs to the two following spaces:*

$$A \otimes_p B \multimap A \otimes_q B \qquad\qquad !_{2^{1/p - 1/q}}(A \otimes_q B) \multimap A \otimes_p B.$$

2. *In particular, when $p = 1$ and $q = 2$, the identity map belongs to:*

$$A \otimes_1 B \multimap A \otimes_2 B \qquad\qquad !_{\sqrt{2}}(A \otimes_2 B) \multimap A \otimes_1 B.$$

Proof. For (1), the fact that the identity belongs to the first space follows from the fact that $d_q(x, y) \leq d_p(x, y)$, by Proposition 3 (Equation (9)). The second claim is derived from Proposition 3 (Equation (9)) in the case $n = 2$.

Remark 1. Proposition 6 allows us to relate different spaces of functions with multiple arguments. For example,

$$(A \otimes_2 B \multimap C) \subseteq (A \otimes_1 B \multimap C) \quad (A \otimes_1 B \multimap C) \subseteq (!_{\sqrt{2}}(A \otimes_2 B) \multimap C).$$

Bunched Fuzz does not currently exploit these inclusions in any significant way, but we could envision extending the system with a notion of subtyping to further simplify the use of multiple product metrics in a single program.

We also have the following result, which is instrumental to prove the soundness of the contraction rule.

Proposition 7. *Let X, Y, Z, W be metric spaces, and $p, q \in \mathbb{R}^{\geq 1}_{\infty}$ with $p \neq \infty$. The canonical isomorphism of sets $(X \times Y) \times (Z \times W) \cong (X \times Z) \times (Y \times W)$, which swaps the second and third components, is a non-expansive function of type $!_{c(p,q)}((X \otimes_q Y) \otimes_p (Z \otimes_q W)) \to (X \otimes_p Z) \otimes_q (Y \otimes_p W)$, where $c(p, q)$ is defined as in Figure 2.*

Proof. First, suppose that $p \leq q$. Then we can write the isomorphism as a composite of the following non-expansive functions:

$$\begin{aligned}
&!_{c(p,q)}((X \otimes_q Y) \otimes_p (Z \otimes_q W) \\
&\to\, !_{c(p,q)}((X \otimes_q Y) \otimes_q (Z \otimes_q W)) && \text{Proposition 6} \\
&\cong\, !_{c(p,q)}((X \otimes_q Z) \otimes_q (Y \otimes_q W)) && \text{assoc., comm. of } \otimes_q \\
&=\, !_{c(p,q)}(X \otimes_q Z) \otimes_q\, !_{c(p,q)}(Y \otimes_q W) && \text{Proposition 4} \\
&= (X \otimes_p Z) \otimes_q (Y \otimes_p W) && \text{Proposition 6.}
\end{aligned}$$

Otherwise, $p > q$, and we reason as follows.

$$\begin{aligned}
&!_{c(p,q)}((X \otimes_q Y) \otimes_p (Z \otimes_q W) \\
&\to\, !_{c(p,q)}((X \otimes_p Y) \otimes_q (Z \otimes_p W)) && \text{Proposition 6} \\
&\cong\, !_{c(p,q)}((X \otimes_p Z) \otimes_p (Y \otimes_p W)) && \text{assoc., comm. of } \otimes_p \\
&= (X \otimes_p Z) \otimes_q (Y \otimes_p W) && \text{Proposition 6.}
\end{aligned}$$

One can then prove the following property:

Proposition 8. *Suppose that we have two bunches $\Gamma \approx \Delta$. The carrier sets of $[\![\Gamma]\!]$ and $[\![\Delta]\!]$ are the same. Moreover, for any p, the diagonal function $\delta(x) = (x, x)$ is a non-expansive function of type $[\![Contr(p, \Gamma, \Delta)]\!] \to [\![\Gamma]\!] \otimes_p [\![\Delta]\!]$.*

Function Types The metric on $\multimap_p$ can be justified by the following result:

Proposition 9. *For every metric space X and every $p \in \mathbb{R}^{\geq 1}_{\infty}$, there is an adjunction of type $(-) \otimes_p X \dashv X \multimap_p (-)$ in* Met *given by currying and uncurrying. (Both constructions on metric spaces are extended to endofunctors on* Met *in the obvious way.)*

Because right adjoints are unique up to isomorphism, this definition is a direct generalization of the metric on functions used in Fuzz [23,4,3], which corresponds to $\multimap_1$.

Theorem 4. *Suppose that A and B are* proper *metric spaces, and let $f, g : A \to B$ be non-expansive. Then $d_{A \multimap_1 B}(f, g) = \sup_x d_B(f(x), g(x))$.*

We conclude with another subtyping result involving function spaces.

Theorem 5. *For all non-expansive functions $f, g \in A \to B$ and $p \geq 1$, we have $d_{A \multimap_1 B}(f, g) \leq d_{A \multimap_p B}(f, g)$. In particular, the identity function is a non-expansive function of type $(A \multimap_p B) \to (A \multimap_1 B)$.*

Probability Distributions Prior work [3] proves that the return and bind operations on probability distributions can be seen as non-expansive functions:

$$\begin{aligned}
\eta &: !_\infty A \to \bigcirc_P A \\
(-)^\dagger(-) &: (!_\infty A \multimap_1 \bigcirc_P B) \otimes_1 \bigcirc_P A \to \bigcirc_P B.
\end{aligned}$$

These properties ensure the soundness of the typing rules for $\bigcirc_P$ in Fuzz, and also in Bunched Fuzz. For $\bigcirc_H$, we can use the following composition principle.

Theorem 6. *The following types are sound for the monadic operations on distributions, seen as non-expansive operations, for any $p \geq 1$:*

$$\begin{aligned} \eta &: !_\infty A \to \bigcirc_H A \\ (-)^\dagger(-) &: (!_\infty A \multimap_p \bigcirc_H B) \otimes_2 \bigcirc_H A \to \bigcirc_H B. \end{aligned}$$

Derivations Finally, a derivation tree builds a function from the context's space to the subject's space. In the following definition, we use the metavariables γ and δ to denote variable assignments—that is, mappings from the variables of environments Γ and Δ to elements of the corresponding metric spaces. We use $\gamma(\delta)$ to represent an assignment in $\llbracket \Gamma(\Delta) \rrbracket$ that is decomposed into two assignments $\gamma(\star)$ and δ corresponding to the $\Gamma(\star)$ and Δ portions. Finally, we use the λ-calculus notation $f\ x$ to denote a function f being applied to the value x.

Definition 1. *Given a derivation π proving $\Gamma \vdash e : \tau$, its interpretation $\llbracket \pi \rrbracket \in \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket$ is given by structural induction on π as follows:*

$$\begin{array}{ll}
\llbracket \mathit{Axiom} \rrbracket \triangleq \lambda x.\ x & \llbracket \mathbb{R}I \rrbracket \triangleq \lambda().\ r \in \mathbb{R} \\
\llbracket \multimap I\ \pi \rrbracket \triangleq \lambda\gamma.\ \lambda x.\ \llbracket \pi \rrbracket\ (\gamma, x) & \llbracket \multimap E\ \pi_1\ \pi_2 \rrbracket \triangleq \lambda(\gamma,\delta).\ \llbracket \pi_2 \rrbracket\ \gamma\ (\llbracket \pi_1 \rrbracket\ \delta) \\
\llbracket 1I \rrbracket \triangleq \lambda().\ () & \llbracket \otimes I\ \pi_1\ \pi_2 \rrbracket \triangleq \lambda(\gamma,\delta).\ (\llbracket \pi_1 \rrbracket\ \gamma), (\llbracket \pi_2 \rrbracket\ \delta) \\
 & \llbracket \otimes E\ \pi_1\ \pi_2 \rrbracket \triangleq \lambda\gamma(\delta).\ \llbracket \pi_2 \rrbracket\ \gamma(\llbracket \pi_1 \rrbracket \delta) \\
\llbracket \oplus_i I\ \pi \rrbracket \triangleq \lambda\gamma.\ \boldsymbol{inj}_i \llbracket \pi \rrbracket\ \gamma & \llbracket \oplus E\ \pi_1\ \pi_2 \rrbracket \triangleq \lambda\delta(\gamma).\ [\llbracket \pi_2 \rrbracket, \llbracket \pi_3 \rrbracket](\delta(\llbracket \pi_1 \rrbracket \gamma)) \\
\llbracket !I\ \pi \rrbracket \triangleq \llbracket \pi \rrbracket & \llbracket !E\ \pi_1\ \pi_2 \rrbracket \triangleq \lambda\ \delta(\gamma).\ \llbracket \pi_2 \rrbracket\ \delta(\llbracket \pi_1 \rrbracket\ \gamma) \\
\llbracket \mathit{Contr}\ \pi \rrbracket \triangleq \lambda\gamma(\delta).\ \llbracket \pi \rrbracket\ \gamma(\delta,\delta) & \llbracket \mathit{Weak}\ \pi \rrbracket \triangleq \lambda\gamma(\delta).\ \llbracket \pi \rrbracket\ \gamma(\ ()\ ) \\
\llbracket \mathit{Exch}\ \pi \rrbracket \triangleq \lambda\gamma'.\llbracket \pi \rrbracket \phi_{\gamma'/\gamma}(\gamma') & \llbracket \mathit{Bind\text{-}P}\ \pi_1\ \pi_2 \rrbracket \triangleq \lambda\gamma'.\ (\llbracket \pi_2 \rrbracket \gamma')^\dagger (\llbracket \pi_1 \rrbracket \gamma') \\
\llbracket \mathit{Return\text{-}P}\ \pi \rrbracket \triangleq \lambda\gamma.\ \eta(\llbracket \pi \rrbracket\ \gamma) &
\end{array}$$

where in $\llbracket \mathit{Exch}\ \pi \rrbracket$, the map $\phi_{\Gamma'/\Gamma}$ is the isomorphism defined by Theorem 3. and for the two last cases see definitions in equations (3) and (4) (Bind-H and Return-H are defined in the same way).

Theorem 7 (Soundness). *Given a derivation π proving $\Gamma \vdash e : \tau$, then $\llbracket \pi \rrbracket$ is a non-expansive function from the space $\llbracket \Gamma \rrbracket$ to the space $\llbracket \tau \rrbracket$.*

5 Examples

We now look at examples of programs that illustrate the use of L^p metrics.

Currying and Uncurrying Let us illustrate the use of higher-order functions with combinators for currying and uncurrying.

$$\begin{aligned} \mathit{curry} &: ((\tau \otimes_p \sigma) \multimap_p \rho) \multimap (\tau \multimap_p \sigma \multimap_p \rho) \\ \mathit{curry}\ f\ x\ y &= f(x, y) \\ \mathit{uncurry} &: (\tau \multimap_p \sigma \multimap_p \rho) \multimap ((\tau \otimes_p \sigma) \multimap_p \rho). \\ \mathit{uncurry}\ f\ z &= \mathbf{let}\ (x, y) = z\ \mathbf{in}\ f\ x\ y \end{aligned}$$

Note that the indices on $\otimes$ and $\multimap$ need to be the same. The reason can be traced back to the $\multimap$ E rule (cf. Figure 3), which uses the $,_p$ connective to eliminate $\multimap_p$ (cf. the currying and uncurrying derivation in the appendix of the full paper for a detailed derivation). If the indices do not agree, currying is not possible; in other words, we cannot in general soundly curry a function of type $\tau \otimes_p \sigma \multimap_q \rho$ to obtain something of type $\tau \multimap_p \sigma \multimap_q \rho$. However, if $q \leq p$, note that it would be possible to soundly view $\tau \otimes_q \sigma$ as a subtype of $\tau \otimes_p \sigma$, thanks to Proposition 6. In this case, we could then convert from $\tau \otimes_p \sigma \multimap_q \rho$ to $\tau \otimes_q \sigma \multimap_q \rho$ (note the variance), and then curry to obtain a function of type $\tau \multimap_q \sigma \multimap_q \rho$.

Precise sensitivity for functions with multiple arguments Another useful feature of Bunched Fuzz is that its contraction rule allows us to split sensitivities more accurately than if we used the contraction rule that is derivable in the original Fuzz. Concretely, suppose that we have a program $\lambda p.\mathbf{let}\ (x, y) = p\ \mathbf{in}\ f(x, y) + g(x, y)$, where f and g have types $f : (!_2\mathbb{R}) \otimes_2 \mathbb{R} \multimap \mathbb{R}$ and $g : \mathbb{R} \otimes_2 (!_2\mathbb{R}) \multimap \mathbb{R}$, and where we have elided the wrapping and unwrapping of ! types, for simplicity.

Let us sketch how this program is typed in Bunched Fuzz. Addition belongs to $\mathbb{R} \otimes_1 \mathbb{R} \multimap \mathbb{R}$, so by Proposition 6 it can also be given the type $!_{\sqrt{2}}(\mathbb{R} \otimes_2 \mathbb{R}) \multimap \mathbb{R}$. Thus, we can build the following derivation for the body of the program:

$$\textsc{Contr}\ \frac{\Gamma \vdash f(x_1, y_1) + g(x_2, y_2) : \mathbb{R}}{[x : \mathbb{R}]_{\sqrt{10}} ,_2 [y : \mathbb{R}]_{\sqrt{10}} \vdash f(x, y) + g(x, y) : \mathbb{R}}$$

where $\Gamma = ([x_1 : \mathbb{R}]_{2\sqrt{2}} ,_2 [y_1 : \mathbb{R}]_{\sqrt{2}}) ,_2 ([x_2 : \mathbb{R}]_{\sqrt{2}} ,_2 [y_2 : \mathbb{R}]_{2\sqrt{2}})$, and where we used contraction twice to merge the xs and ys. Note that $||(2\sqrt{2}, \sqrt{2})||_2 = \sqrt{8+2} = \sqrt{10}$, which is why the final sensitivities have this form. By contrast, consider how we might attempt to type this program directly in the original Fuzz. Let us assume that we are working in an extension of Fuzz with types for expressing the domains of f and g, similarly to the L^2 vector types of Duet [20]. Moreover, let us assume that we have coercion functions that allow us to cast from $(!_2\mathbb{R}) \otimes_2 (!_2\mathbb{R})$ to $(!_2\mathbb{R}) \otimes_2 \mathbb{R}$ and $\mathbb{R} \otimes_2 (!_2\mathbb{R})$. If we have a pair $p :!_2((!_2\mathbb{R}) \otimes_2 (!_2\mathbb{R}))$, we can split its sensitivity to call f and g and then combine their results with addition. However, this type is equivalent to $!_4(\mathbb{R} \otimes_2 \mathbb{R})$, which means that the program was given a worse sensitivity (since $\sqrt{10} < 4$). Of course, it would also have been possible to extend Fuzz with a series of primitives that implement precisely the management of sensitivities performed by bunches. However, here this low-level reasoning is handled directly by the type system.

Programming with matrices The Duet language [20] provides several matrix types with the L^1, L^2, or L^∞ metrics, along with primitive functions for manipulating them. In Bunched Fuzz, these types can be defined directly as follows: $\mathbb{M}_p[m, n] = \otimes_1^m \otimes_p^n \mathbb{R}$. Following Duet, we use the L^1 distance to combine the rows and the L^p distance to combine the columns. One advantage of having types for matrices defined in terms of more basic constructs is that we can program functions for manipulating them directly, without resorting to separate primitives. For example, we can define the following terms in the language:

$$\begin{aligned} \mathit{addrow} &: \mathbb{M}_p[1,n] \otimes_1 \mathbb{M}_p[m,n] \multimap \mathbb{M}_p[m+1,n] \\ \mathit{addcolumn} &: \mathbb{M}_1[1,m] \otimes_1 \mathbb{M}_1[m,n] \multimap \mathbb{M}_1[m,n+1] \\ \mathit{addition} &: \mathbb{M}_1[m,n] \otimes_1 \mathbb{M}_1[m,n] \multimap \mathbb{M}_1[m,n]. \end{aligned}$$

The first program, *addrow*, appends a vector, represented as a $1 \times n$ matrix, to the first row of a $m \times n$ matrix. The second program, *addcolumn*, is similar, but appends the vector as a column rather than a row. Because of that, it is restricted to L^1 matrices. Finally, the last program, *addition*, adds the elements of two matrices pointwise.

Vector addition over sets Let us now show an example of a Fuzz term for which using L^p metrics allows to obtain a finer sensitivity analysis. We consider sets of vectors in $\mathbb{R}^d$ and the function *vectorSum* which, given such a set, returns the vectorial sum of its elements. In Fuzz, this function can be defined via a summation primitive $\mathit{sum} : !_\infty(!_\infty \tau \multimap \mathbb{R}) \multimap \mathit{set}\,\tau \multimap \mathbb{R}$, which adds up the results of applying a function to each element of a set [23]. The definition is:

$$\begin{gathered} \mathit{vectorSum} : !_d\, \mathit{set}(\otimes_1^d \mathbb{R}) \multimap_1 \otimes_1^d \mathbb{R} \\ \mathit{vectorSum}\ s = (\mathit{sum}\ \pi_1\ s, \ldots, \mathit{sum}\ \pi_d\ s). \end{gathered}$$

Here, $\pi_i : \otimes_1^d \mathbb{R} \multimap \mathbb{R}$ denotes the i-th projection, which can be defined by destructing a product. Set types in Fuzz are equipped with the Hamming metric [23], where the distance between two sets is the number of elements by which they differ. Note that, to ensure that *sum* has bounded sensitivity, we need to clip the results of its function argument to the interval $[-1, 1]$. Fuzz infers a sensitivity of d for this function because its argument is used with sensitivity 1 in each component of the tuple. In Bunched Fuzz, we can define the same function as above, but we also have the option of using a different L^p distance to define *vectorSum*, which leads to the type $!_{d^{1/p}}\, \mathit{set}(\otimes_p^d \mathbb{R}) \multimap \otimes_p^d \mathbb{R}$, with a sensitivity of $d^{1/p}$. For the sake of readability, we'll show how this term is typed in the case $d = 2$. By typing each term $(\mathit{sum}\ \pi_i\ z_i)$ and applying $(\otimes I)$ we get:

$$[z_1 : \mathit{set}(\mathbb{R} \otimes_p \mathbb{R})]_1 \,,_p [z_2 : \mathit{set}(\mathbb{R} \otimes_p \mathbb{R})]_1 \vdash (\mathit{sum}\ \pi_1\ z_1, \mathit{sum}\ \pi_2\ z_2) : \mathbb{R} \otimes_p \mathbb{R}.$$

By applying contraction we get: $[z : \mathit{set}(\mathbb{R} \otimes_p \mathbb{R})]_{2^{1/p}} \vdash (\mathit{sum}\ \pi_1\ z, \mathit{sum}\ \pi_2\ z) : \mathbb{R} \otimes_p \mathbb{R}$. The claimed type is finally obtained by $(!E)$ and $(\multimap I)$.

Computing distances Suppose that the type X denotes a proper metric space (that is, where the triangle inequality holds). Then we can incorporate its distance function in Bunched Fuzz with the type $X \otimes_1 X \multimap \mathbb{R}$. Indeed, let x, x', y and y' be arbitrary elements of X. Then

$$\begin{aligned} d_X(x,y) - d_X(x',y') &\le d_X(x,x') + d_X(x',y') + d_X(y',y) - d_X(x',y') \\ &= d_X(x,x') + d_X(y,y') = d_1((x,y),(x',y')). \end{aligned}$$

By symmetry, we also know that $d_X(x', y') - d_X(x, y) \leq d_1((x, y), (x', y'))$. Combined, these two facts show

$$d_{\mathbb{R}}(d_X(x, y), d_X(x', y')) = |d_X(x, y) - d_X(x', y')| \leq d_1((x, y), (x', y')),$$

which proves that d_X is indeed a non-expansive function.

Calibrating noise to L^p distance Hardt and Talwar [17] have proposed a generalization of the Laplace mechanism, called the K-norm mechanism, to create a differentially private variant of a database query $f : \mathtt{db} \to \mathbb{R}^d$. The difference is that the amount of noise added is calibrated to the sensitivity of f measured with the K norm, as opposed to the L^1 distance used in the original Laplace mechanism. When K corresponds to the L^p norm, we will call this the L^p-mechanism, following Awan and Slavkovich [1].

Definition 2. *Given $f : \mathtt{db} \to \mathbb{R}^d$ with L^p sensitivity s and $\epsilon > 0$, the L^p-mechanism is a mechanism that, given a database $D \in \mathtt{db}$, returns a probability distribution over $y \in \mathbb{R}^d$ with density given by:*

$$\frac{\exp(\frac{-\epsilon||f(D)-y||_p}{2s})}{\int \exp(\frac{-\epsilon||f(D)-y||_p}{2s})dy}$$

This mechanism returns with high probability (which depends on ϵ and on the sensitivity s) a vector $y \in \mathbb{R}^d$ which is close to $f(D)$ in L^p distance. Such a mechanism can be easily integrated in Bunched Fuzz through a primitive:

$$\mathtt{LpMech} : !_\infty(!_s\mathtt{dB} \multimap \otimes_p^d\mathbb{R}) \multimap !_\epsilon\mathtt{dB} \multimap \bigcirc_P(\otimes_p^d\mathbb{R})$$

(Strictly speaking, we would need some discretized version of the above distribution to incorporate the mechanism in Bunched Fuzz, but we'll ignore this issue in what follows.) The fact that `LpMech` satisfies ϵ-differential privacy follows from the fact that this mechanism is an instance of the *exponential mechanism* [18], a basic building block of differential privacy. It is based on a scoring function assigning a score to every pair consisting of a database and a potential output, and it attempts to return an output with approximately maximal score, given the input database. As shown by Gaboardi et al. [13], the exponential mechanism can be added as a primitive to Fuzz with type:

$$\mathtt{expmech} : !_\infty\, set(\mathcal{O}) \multimap !_\infty(!_\infty\mathcal{O} \multimap !_s\mathtt{dB} \multimap \mathbb{R}) \multimap !_\epsilon\mathtt{dB} \multimap \bigcirc_P\mathcal{O},$$

where $\mathcal{O}$ is the type of outputs. The function `LpMech` is an instance of the exponential mechanism where $\mathcal{O}$ is $\otimes_p^d\mathbb{R}$ and the score is $\lambda y \lambda D.||f(D) - y||_p$.

To define the L^p mechanism with this recipe, we need to reason about the sensitivity of this scoring function. In Fuzz, this would not be possible, since the language does not support reasoning about the sensitivity of f measured in the L^p distance. In Bunched Fuzz, however, this can be done easily. Below, we will see an example (Gradient descent) of how the L^p mechanism can lead to a finer privacy guarantee.

Gradient descent Let us now give an example where we use the L^p mechanism. An example of differentially private gradient descent example with linear model in Fuzz was given in [25] (see Sect. 4.1, 4.2 and Fig. 6 p. 16, Fig. 8 p.19). This algorithm proceeds by iteration. Actually it was given for an extended language called Adaptative Fuzz, but the code already gives an algorithm in (plain) Fuzz. We refer the reader to this reference for the description of all functions, and here we will only describe how one can adapt the algorithm to Bunched Fuzz.

Given a set of n records $x_i \in \mathbb{R}^d$, each with a *label* $y_i \in \mathbb{R}$, the goal is to find a parameter vector $\theta \in \mathbb{R}^d$ that minimizes the difference between the labels and their *estimates*, where the estimate of a label y_i is the inner product $\langle x_i, \theta \rangle$. That is, the goal is to minimize the loss function $L(\theta, (x, y)) = \frac{1}{n} \cdot \Sigma_{i=1}^{n}(\langle x_i, \theta\rangle - y_i)^2$. The algorithm starts with an initial parameter vector $(0, \ldots, 0)$ and it iteratively produces successive θ vectors until a termination condition is reached.

The Fuzz program uses the data-type *bag* τ representing bags or multisets over τ. A *bagmap* primitive is given for it. The type I is the unit interval $[0, 1]$. The main function is called *updateParameter* and updates one component of the model θ; it is computed in the following way:

- with the function $calcGrad : \mathtt{db} \rightarrow \mathbb{R}$, compute a component $(\nabla L(\theta, (x, y)))_j$ of the $\mathbb{R}^d$ vector $\nabla L(\theta, (x, y))$ [9].
- then Laplacian noise is postcomposed with *calcGrad* in the *updateParameter* function. This uses a privacy budget of 2ϵ. It has to be done for each one of the d components of $\nabla L(\theta, (x, y))$, thus on the whole, for one step, a privacy budget of $2d\epsilon$.
- The iterative procedure of gradient descent is given by the function *gradient* in Fig. 8 p. 19 of [25]. We forget here about the adaptative aspect and just consider iteration with a given number n of steps. In this case by applying n times *updateParameter* one gets a privacy budget of $2dn\epsilon$.

We modify the program as follows to check it in Bunched Fuzz and use the L^p-mechanism. Instead of computing over $\mathbb{R}$ we want to compute over $\otimes_p^d \mathbb{R}$ for a given $p \geq 1$, so $\mathbb{R}^d$ equipped with L^p distance. The records x_i are in $\otimes_p^d I$ and the labels y_i in I. The database type is $\mathtt{dB} = bag\ (I \otimes_p (\otimes_p^d I))$. The distance between two bags in $\mathtt{dB}$ is the number of elements by which they differ.

We assume a primitive *bagVectorSum* with type $!_{d^{1/p}} bag\ (\otimes_p^d I) \multimap \otimes_p^d \mathbb{R}$ (it could be defined as the *vectorSum* defined above for sets, using a *sum* primitive for bags). Given a bag m, $(bagVectorSum\ m)$ returns the vectorial sum of all elements of m. We can check that the sensitivity of *bagVectorSum* is indeed $d^{1/p}$ because given two bags m and m' that are at distance 1, if we denote by u the vector by which they differ, we have:

$$d_{(\otimes_p^d \mathbb{R})}(bagVectorSum(m), bagVectorSum(m')) = ||u||_p \quad \leq (\Sigma_{j=1}^{d} 1)^{1/p} = d^{1/p}$$

By adapting the *calcGrad* Fuzz term of [25] using *bagVectorSum* we obtain a term *VectcalcGrad* with the Bunched Fuzz type $!_\infty \otimes_p^d \mathbb{R} \multimap !_{d^{1/p}} \mathtt{db} \multimap \otimes_p^d \mathbb{R}$.

[9] Actually *calcGrad* computes $(\nabla L(\theta, (x, y)))_j$ up to a multiplicative constant, 2/n, which is mutliplied afterwards in the *updateParameter* function.

Given a vector θ and a database (y, x), $VectcalcGrad$ computes the updated vector θ'. Finally we define the term $updateVector$ by adding noise to $VectcalcGrad$ using the the L^p-mechanism. Recall the type of `LpMech`: $!_\infty(!_s\mathtt{db} \multimap \otimes_p^d \mathbb{R}) \multimap\ !_\epsilon\mathtt{db} \multimap \bigcirc_P(\otimes_p^d\mathbb{R})$. We define $updateVector$ and obtain its type as follows:

$$updateVector = \lambda\theta.(\mathtt{LpMech}\ (VectcalcGrad\ \theta)) : !_\infty \otimes_p^d \mathbb{R} \multimap !_\epsilon\mathtt{db} \multimap \bigcirc_P(\otimes_p^d\mathbb{R})$$

By iterating $updateVector$ n times one obtains a privacy budget of $n\epsilon$.

6 Implementation

To experiment with the Bunched Fuzz design, we implemented a prototype for a fragment of the system based on DFuzz [13,2].[10] The type-checker generates a set of numeric constraints that serve as verification conditions to guarantee a valid typing. The implementation required adapting some of the current rules to an algorithmic formulation (found in the full version). In addition to the modifications introduced in the DFuzz type checker compared to its original version [13,2], we also made the following changes and simplifications:

- We did not include explicit contraction and weakening rules. Instead, the rules are combined with those for checking other syntactic constructs. To do away with an explicit contraction rule, in rules that have multiple antecedents, such as the $\otimes$I rule, we used the *Contr* operator to combine the antecedents' environments, rather than using the p-concatenation operator for bunches.
- We did not include the rules for checking probabilistic programs with the Hellinger distance.
- Bound variables are always added at the top of the current environment, as in the $\multimap$I rule of the original rules; it is not possible to introduce new variables arbitrarily deep in the environment.

While, strictly speaking, the resulting system is incomplete with respect to the rules presented here, it is powerful enough to check an implementation of K-means that generalizes a previous version implemented for Fuzz [23]. On the other hand, because our implementation is based on the one of DFuzz, which features dependent types, we allow functions that are polymorphic on types, sizes and p parameters, which allows us to infer sensitivity information that depends on run-time sizes.

7 Related Work

Bunched Fuzz is inspired by BI, the logic of bunched implications [22], which has two connectives for combining contexts. Categorically, one of these connectives corresponds to a Cartesian product, whereas the other corresponds to a

[10] https://github.com/junewunder/bunched-fuzz

monoidal, or tensor product. While related to linear logic, the presence of the two context connectives allows BI to derive some properties that are not valid in linear logic. For example, the cartesian product does not distribute over sums in linear logic but it does distribute over sums in BI.

We have shown how the rules for such type systems are reminiscent of the ones used in type systems for the calcuclus of bunched implications [21], and for reasoning about categorical grammars [19]. Specifically, O'Hearn introduces a type system with two products and two arrows [21]. Typing environments are bunches of variable assignments with two constructors, corresponding to the two products. Our work can be seen as a generalization of O'Hearn's work to handle multiple products and to reason about program sensitivity.

Moot and Retoré [19, Chapter 5] introduce the multimodal Lambek calculus, which extends the non-associative Lambek calculus, a classical tool for describing categorical grammars. This generalization uses an indexed family of connectives and trees to represent environments. The main differences with our work are: our indexed products are associative and commutative, while theirs are not; our type system is affine; our type system includes a monad for probabilities which does not have a correspondent construction in their logic; our type system also possesses the graded comonad $!_s$ corresponding to the ! modality of linear logic, the interaction between this comonad and the bunches is non-trivial and it requires us to explicitly define a notion of contraction. Besides the fact that the main properties we study, metric interpretation and program sensitivity, are very different from the ones studied by the above authors, there are some striking similarities between the two systems.

A recent work by Bao et al. [5] introduced a novel bunched logic with indexed products and magic wands with a preorder between the indices. This logic is used as the assertion logic of a separation logic introduced to reason about negative dependence between random variables. The connectives studied in this work share some similarities with the ones we study here and it would be interesting to investigate further the similarities, especially from a model-theoretic perspective.

Because contexts in the original Fuzz type system are biased towards the L^1 distance, it is not obvious how Fuzz could express the composition principles of the Hellinger distance. Recent work showed how this could be amended via a *path construction* that recasts relational program properties as sensitivity properties [3]. Roughly speaking, instead of working directly with the Hellinger distance d_H, the authors consider a family of relations $R_\alpha = \{(\mu_1, \mu_2) \mid d_H(\mu_1, \mu_2) \leq \alpha\}$. Such a relation induces another metric on distributions, $d_{\alpha,H}$, where the distance between two distributions is the length of the shortest path connecting them in the graph corresponding to R_α. This allows them to express the composition principles of the Hellinger distance directly in the Fuzz type system, albeit at a cost: the type constructor for probability distributions is graded by the distance bound α. Thus, the sensitivity information of a randomized algorithm with respect to the Hellinger distance must also be encoded in the codomain of the function, as opposed to using just its domain, as done for the original privacy metric of Fuzz. By contrast, Bunched Fuzz does not require the grading α because it can express the composition principle of the Hellinger distance directly, thanks to the use of the L^2 distance on bunches.

Duet [20] can be seen as an extension of Fuzz to deal with more general privacy distances. It consists of a two-layer language: a sensitivity language and a privacy language. The sensitivity language is very similar to Fuzz. However, it also contains some basic primitives to manage vectors and matrices. As in Fuzz, the vector types come with multiple distances but differently from Fuzz, Duet also uses the L^2 distance. The main reason for this is that Duet also supports the Gaussian mechanism which calibrates the noise to the L^2 sensitivity of the function. Our work is inspired by this aspect of Duet, but it goes beyond it by giving a logical foundation to L^p vector distances. Another language inspired by Fuzz is the recently proposed Jazz [24]. Like Duet, this language has two products and primitives tailored to the L^2 sensitivity of functions for the Gaussian mechanism. Interestingly, this language uses contextual information to achieve more precise bounds on the sensitivities. The semantics of Jazz is different from the metric semantics we study here; however, it would be interesting to explore whether a similar contextual approach could be also used in a metric setting.

8 Conclusion and Future work

In this work we have introduced Bunched Fuzz, a type system for reasoning about program sensitivity in the style of Fuzz [23]. Bunched Fuzz extends the type theory of Fuzz by considering new type constructors for L^p distances and bunches to manage different products in typing environments. We have shown how this type system supports reasoning about both deterministic and probabilistic programs.

There are at least two directions that we would like to explore in future works. On the one hand, we would like to understand if the typing rules we introduced here could be of more general use in the setting of probabilistic programs. We have already discussed the usefulness for other directions in the deterministic case [19]. One way to approach this problem could be by looking at the family of products recently identified in [5]. These products give a model for a logic to reason about negative dependence between probabilistic variables. It would be interesting to see if the properties of these products match the one we have here.

On the other hand, we would like to understand if Bunched Fuzz can be used to reason about more general examples in differential privacy. One way to approach this problem could be to consider examples based on the use of Hellinger distance that have been studied in the literature on probabilistic inference [6].

Acknowledgements This material is based upon work supported by the NSF under Grant No. 1845803 and 2040249. The third author was partially supported by the french Program “Investissements d’avenir” (I-ULNE SITE / ANR-16-IDEX-0004 ULNE) managed by the National Research Agency.

References

1. Awan, J., Slavkovic, A.: Structure and sensitivity in differential privacy: Comparing k-norm mechanisms. Journal of the American Statistical Association (2020). https://doi.org/10.1080/01621459.2020.1773831, https://par.nsf.gov/biblio/10183971
2. Azevedo de Amorim, A., Gaboardi, M., Arias, E.J.G., Hsu, J.: Really natural linear indexed type checking. In: Tobin-Hochstadt, S. (ed.) Proceedings of the 26th 2014 International Symposium on Implementation and Application of Functional Languages, IFL '14, Boston, MA, USA, October 1-3, 2014. pp. 5:1–5:12. ACM (2014). https://doi.org/10.1145/2746325.2746335
3. Azevedo de Amorim, A., Gaboardi, M., Hsu, J., Katsumata, S.: Probabilistic relational reasoning via metrics. In: 34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019. pp. 1–19. IEEE (2019). https://doi.org/10.1109/LICS.2019.8785715
4. Azevedo de Amorim, A., Gaboardi, M., Hsu, J., Katsumata, S., Cherigui, I.: A semantic account of metric preservation. In: POPL 2017. ACM (2017), http://dl.acm.org/citation.cfm?id=3009890
5. Bao, J., Gaboardi, M., Hsu, J., Tassarotti, J.: A separation logic for negative dependence. Proc. ACM Program. Lang. **6**(POPL) (jan 2022). https://doi.org/10.1145/3498719
6. Barthe, G., Farina, G.P., Gaboardi, M., Arias, E.J.G., Gordon, A., Hsu, J., Strub, P.: Differentially private bayesian programming. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 68–79. ACM (2016). https://doi.org/10.1145/2976749.2978371
7. Bousquet, O., Elisseeff, A.: Stability and generalization. J. Mach. Learn. Res. **2**, 499–526 (2002), http://jmlr.org/papers/v2/bousquet02a.html
8. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge University Press (March 2004)
9. Chaudhuri, S., Gulwani, S., Lublinerman, R., NavidPour, S.: Proving programs robust. In: Gyimóthy, T., Zeller, A. (eds.) SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011. pp. 102–112. ACM (2011). https://doi.org/10.1145/2025113.2025131
10. Csiszár, I., Shields, P.: Information theory and statistics: A tutorial. Foundations and Trends® in Communications and Information Theory **1**(4), 417–528 (2004). https://doi.org/10.1561/0100000004, http://dx.doi.org/10.1561/0100000004
11. Dwork, C., McSherry, F., Nissim, K., Smith, A.D.: Calibrating noise to sensitivity in private data analysis. In: Halevi, S., Rabin, T. (eds.) Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings. Lecture Notes in Computer Science, vol. 3876, pp. 265–284. Springer (2006). https://doi.org/10.1007/11681878_14
12. Dwork, C., Roth, A.: The algorithmic foundations of differential privacy. Found. Trends Theor. Comput. Sci. **9**(3-4), 211–407 (2014). https://doi.org/10.1561/0400000042

13. Gaboardi, M., Haeberlen, A., Hsu, J., Narayan, A., Pierce, B.C.: Linear dependent types for differential privacy. In: POPL '13. ACM (2013). https://doi.org/10.1145/2429069.2429113

14. Girard, J.: Linear logic. Theor. Comput. Sci. **50**, 1–102 (1987). https://doi.org/10.1016/0304-3975(87)90045-4

15. Gonin, R., Money, A.H.: Nonlinear Lp-Norm Estimation. Marcel Dekker, Inc., USA (1989)

16. Haeberlen, A., Pierce, B.C., Narayan, A.: Differential privacy under fire. In: 20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings. USENIX Association (2011), http://static.usenix.org/events/sec11/tech/full_papers/Haeberlen.pdf

17. Hardt, M., Talwar, K.: On the geometry of differential privacy. In: Schulman, L.J. (ed.) Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010. pp. 705–714. ACM (2010). https://doi.org/10.1145/1806689.1806786

18. McSherry, F., Talwar, K.: Mechanism design via differential privacy. In: 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Providence, RI, USA, Proceedings. pp. 94–103. IEEE Computer Society (2007). https://doi.org/10.1109/FOCS.2007.41

19. Moot, R., Retoré, C.: The Logic of Categorial Grammars - A Deductive Account of Natural Language Syntax and Semantics, Lecture Notes in Computer Science, vol. 6850. Springer (2012). https://doi.org/10.1007/978-3-642-31555-8

20. Near, J.P., Darais, D., Abuah, C., Stevens, T., Gaddamadugu, P., Wang, L., Somani, N., Zhang, M., Sharma, N., Shan, A., Song, D.: Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. Proc. ACM Program. Lang. **3**(OOPSLA) (2019). https://doi.org/10.1145/3360598

21. O'Hearn, P.W.: On bunched typing. J. Funct. Program. **13**(4), 747–796 (2003). https://doi.org/10.1017/S0956796802004495

22. O'Hearn, P.W., Pym, D.J.: The logic of bunched implications. Bull. Symb. Log. **5**(2) (1999). https://doi.org/10.2307/421090

23. Reed, J., Pierce, B.C.: Distance makes the types grow stronger: a calculus for differential privacy. In: ICFP 2010. ACM (2010). https://doi.org/10.1145/1863543.1863568

24. Toro, M., Darais, D., Abuah, C., Near, J., Olmedo, F., Tanter, É.: Contextual linear types for differential privacy. CoRR **abs/2010.11342** (2020), https://arxiv.org/abs/2010.11342

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

26. june wunder, Azevedo de Amorim, A., Baillot, P., Gaboardi, M.: Bunched fuzz: Sensitivity for vector metrics. CoRR **abs/2202.01901** (2022), https://arxiv.org/abs/2202.01901