Linear Dependent Types for Differential Privacy

Marco Gaboardi[⋆‡] Andreas Haeberlen[⋆] Justin Hsu[⋆] Arjun Narayan[⋆] Benjamin C. Pierce[⋆]

[⋆]University of Pennsylvania [‡]Università di Bologna–INRIA EPI Focus

Abstract

Differential privacy offers a way to answer queries about sensitive information while providing strong, provable privacy guarantees, ensuring that the presence or absence of a single individual in the database has a negligible statistical effect on the query's result. Proving that a given query has this property involves establishing a bound on the query's *sensitivity*—how much its result can change when a single record is added or removed.

A variety of tools have been developed for certifying that a given query is differentially private. In one approach, Reed and Pierce [34] proposed a functional programming language, *Fuzz*, for writing differentially private queries. *Fuzz* uses linear types to track sensitivity and a probability monad to express randomized computation; it guarantees that any program with a certain type is differentially private. *Fuzz* can successfully verify many useful queries. However, it fails when the sensitivity analysis depends on values that are not known statically.

We present *DFuzz*, an extension of *Fuzz* with a combination of linear indexed types and lightweight dependent types. This combination allows a richer sensitivity analysis that is able to certify a larger class of queries as differentially private, including ones whose sensitivity depends on runtime information. As in *Fuzz*, the differential privacy guarantee follows directly from the soundness theorem of the type system. We demonstrate the enhanced expressivity of *DFuzz* by certifying differential privacy for a broad class of iterative algorithms that could not be typed previously.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized application languages; F.3.3 [*Theory of computation*]: Studies of Program Constructs—Type structure

General Terms Design, Languages, Theory

Keywords differential privacy, type systems, linear logic, dependent types

1. Introduction

An enormous amount of data accumulates in databases every day—travel reservations, hospital records, location data, etc. This information could potentially be very valuable, e.g., for scientific and medical research, but much of it cannot be safely released due to privacy concerns. Moreover, aggregation and anonymization are not sufficient to safeguard privacy: recent experience with the Netflix prize [30], for example, has shown how easy it is to leak sensitive information accidentally, even when the data is carefully sanitized before it is released.

Differential privacy [4, 12, 13] is a promising approach to this problem: it offers strong statistical privacy guarantees for certain types of queries, even in worst-case scenarios. Intuitively, this is accomplished by a) admitting only queries whose result does not depend too much on the data of any single individual, and b) adding some random noise to the result of each query. Thus, if we pick any individual I and remove all of I's data from the database before answering a query, the probability that the result is any given value v remains almost the same. This limits the amount of information that can be learned about a single individual by observing the result of the query.

Many specific queries have been shown to be differentially private, including machine learning algorithms such as empirical risk minimization [5] and k-means, combinatorial optimization algorithms such as vertex cover and k-medians [18], and many others. But checking that a given query is differentially private can be both tedious and rather subtle. The key challenge is to prove an upper bound on the query's *sensitivity*, i.e., the maximum change in the query's output that can result from changing the data of a single individual. (Briefly, higher-sensitivity queries require more noise to maintain privacy.) Since most data analysts are not experts in differential privacy, they cannot benefit from its strong guarantees unless they have access to suitable tools.

The approach we focus on is to provide analysts with a *programming language for differentially private queries*: the analyst can formulate queries in this language and then submit them to a special compiler, which determines their "privacy cost" and rejects them if this cost exceeds a budget that has been specified by the analyst. This approach is attractive because differential privacy is compositional; for instance, the privacy cost of a sequence of differentially private computations is simply the sum of the individual costs. Thus, we can reason about large, complex queries by manually inspecting a few simple primitives and then suitably composing the analysis results of larger and larger subqueries. This is the basis of previous systems like *PINQ* [26], which provides a SQL-like language, *Airavat* [36], which implements a MapReduce framework, and *Fuzz* [20, 33, 34], a higher-order functional language.

The analysis in Fuzz is based on a type system [33, 34] that certifies queries as differentially private via two components: numeric annotations at the type level to describe the sensitivity of functions and a probability monad to represent probabilistic computations. *Fuzz* can certify many useful queries, but it fails for other important kinds of queries, including some fairly simple ones. For instance, iterative algorithms such as k-means can only be analyzed

POPL'13, January 23–25, 2013, Rome, Italy.

when the number of iterations is a constant, and there are other instances (such as a program that computes a cumulative distribution function for an arbitrary list of cutoffs) where even the *type* of the program cannot be expressed in *Fuzz*! *Fuzz* fails for these programs because their overall sensitivities are not simply a static composition of the sensitivities of subprograms; rather, they depend on some input data, such as the number of iterations or the list of numeric cutoffs. Numeric sensitivity annotations are not sufficient to express such *dependencies* between input data and sensitivity; however, as we show in this paper, these dependencies can be expressed with dependent types.

Dependent types enable static reasoning about information that will be available to a program only at runtime. But we must make a choice at the outset regarding the complexity of the dependencies we wish to support: richer dependencies would expand the range of queries that can be certified as differentially private, but systems with rich dependent types tend to require extensive program annotations, which would make *DFuzz* more difficult to use by non-experts. At the extreme end of this spectrum would be a system like CertiPriv [3], which can certify a broad range of differentially private queries but requires the programmer to supply most of the proof. To preserve our goal of usability by non-experts, we instead work near the other end, choosing a *lightweight* form of dependent types that requires few annotations but still yields a powerful analysis.

We introduce a language called *DFuzz*—*D*ependent *Fuzz*—that combines a rich form of type-level sensitivity annotations with lightweight dependent types. The sensitivity annotations we consider are arithmetic expressions over real numbers. Our dependent types are a simple form of *indexed types*, where indices describe the size of data types and provide the programmer with dependent pattern matching in the style of Dependent ML [38]. Sensitivity annotations and dependent types combine well, and the resulting language possesses strong metatheoretic properties. Most importantly, our type system (like that of *Fuzz*) natively captures the concept of differential privacy, since *DFuzz* enjoys an extension of the metric preservation property of [33, 34]. To demonstrate the capabilities of *DFuzz*, we show that it can certify several examples from the differential privacy literature quite naturally, including Iterative Database Construction [19], k-means [4], k-medians [18], and the Cumulative Distribution Function [27].

In summary, we offer the following contributions:

- *DFuzz*, a core calculus for differential privacy that combines the earlier sensitivity analysis of Reed and Pierce [34] with lightweight dependent types inspired by Dependent ML [38] (Section 3);
- the fundamental metatheory for *DFuzz*, including an adaptation of the usual basic metatheory—substitution, preservation and progress—as well as a generalization of the metric preservation theorem from [33, 34] (Sections 4 and 5); and
- four example programs that express algorithms from the differential privacy literature and can be successfully certified by *DFuzz* (Section 6).

We discuss related work in Section 7 and future research directions in Section 8.

2. Background and Overview

Differential privacy We begin by reviewing some basic definitions. Suppose the private data is collected in a database that contains rows of the same type, and each individual's data is in a single row. Let $\mathbf{db}$ be the type of such databases, and assume for the moment that we are interested in real-valued queries. Then the key definition of differential privacy (based on [13]) is:

2.1 Definition: A randomized function $f : \mathbf{db} \to \mathbb{R}$ is ϵ-*differentially private* if, for all possible sets of outputs $S \subseteq \mathbb{R}$ and for any two databases b, b' that differ in only one row,

$$\mathtt{Pr}[f(b) \in S] \leq e^{\epsilon} \cdot \mathtt{Pr}[f(b') \in S].$$

Intuitively, this means that the presence or absence of one individual's data has only a small effect on the distribution of f's (randomized) outputs. Here ϵ is a privacy parameter; the larger ϵ is, the more information about individuals is potentially revealed by the result. For small ϵ, the factor e^{ϵ} can be thought of as $1 + \epsilon$. We will often refer to ϵ as the *privacy cost* of running f.

We can extend the definition to data types other than $\mathbf{db}$ and $\mathbb{R}$ if we assign to each data type a *metric* that measures the distance between values. We write $\vdash v \sim_r v' : \tau$ to indicate that two values v and v' of type τ are at most r apart. Thus we obtain:

2.2 Definition: A randomized function $q : \tau \to \sigma$ is ϵ-differentially private if, for all sets S of closed values of type σ and for all $v, v' : \tau$ such that $\vdash v \sim_r v' : \tau$, we have:

$$Pr[q(v) \in S] \;\leq\; e^{\epsilon r} Pr[q(v') \in S]$$

To determine whether a given query has this property, the following definition is useful:

2.3 Definition: A function $f : \tau \to \sigma$ is c-*sensitive* for $c \in \mathbb{R}^{\geq 0}$ if, for all $v, v' : \tau$ with $\vdash v \sim_r v' : \tau$, we have $\vdash f(v) \sim_{c \cdot r} f(v') : \sigma$.

In other words, a c-sensitive function magnifies changes in its inputs by at most a factor of c. When a function q is not c-sensitive for any $c \in \mathbb{R}^{\geq 0}$ in the sense of the above definition, we will refer to it as ∞-sensitive. Real-valued functions with limited sensitivity can be converted into ϵ-differentially private queries using the *Laplace mechanism* (introduced in [13]). Here, we write $\mathcal{L}_{\beta}$ to denote the Laplace distribution with scaling parameter β.

2.4 Proposition: Let $f : \mathbf{db} \to \mathbb{R}$ be a c-sensitive function, and let $q : \mathbf{db} \to \mathbb{R}$ be the randomized function $q = \lambda b.\ f(b) + N$, where N is a random variable distributed according to $\mathcal{L}_{c/\epsilon}$. Then q is ϵ-differentially private.

In other words, the Laplace mechanism converts f into a randomized function q by adding noise from the Laplace distribution. Note that the parameter of the distribution—the 'magnitude' of the noise—depends on both c and ϵ: the stronger the privacy requirement (smaller ϵ) and the higher the sensitivity of f (larger c), the more noise must be added to f's result to preserve privacy.

Linear types for differential privacy The key idea behind static type systems for differential privacy is to provide a compositional analysis that tracks both the privacy cost ϵ and the sensitivity c of a function. For context, we sketch the analysis in *Fuzz* [33, 34], on which our work is based.

Fuzz has a *linear type system*: functions can have types of the form $!_r\sigma \multimap \tau$, where the *modality* $!_r$ is annotated with a numeric upper bound r on the function's sensitivity. For instance, the term $\lambda x.(2 \cdot x)$ can be given the type $!_2\mathbb{R} \multimap \mathbb{R}$ to express the fact that $f(x) = 2 \cdot x$ is 2-sensitive in its argument x.[1] *Fuzz* checks these types with standard typing judgments of the form $\Gamma \vdash e : \sigma$, where the type environment Γ additionally contains hypotheses about sensitivity. For instance, a typing judgment $x : !_c\sigma \vdash e : \tau$ means that e can be given type τ if e is at most c-sensitive in the free variable x.

[1] In [33, 34] the symbol ! appears both in types (as here) and in terms (to help guide typechecking algorithms). Here, for simplicity, we elide uses in terms.

As an illustration, consider the slightly more complex program $(\lambda z.3 \cdot z + 8)(5 \cdot x)$ It is easy to see that $x : !_5\mathbb{R} \vdash 5 \cdot x : \mathbb{R}$ and $\vdash \lambda z.3 \cdot z + 8 : !_3\mathbb{R} \multimap \mathbb{R}$, and *Fuzz* thus uses the typing rule

$$\frac{\Gamma \vdash e : !_r\tau \multimap \sigma \quad \Delta \vdash g : \tau}{\Gamma + r\Delta \vdash e\ g : \sigma}$$

to infer $x : !_{15}\mathbb{R} \vdash (\lambda z.3 \cdot z + 8)(5 \cdot x) : \mathbb{R}$. This reflects the fact that $3 \cdot (5 \cdot x) + 8 = 15 \cdot x + 8$ is 15-sensitive in x. To turn this (deterministic) program into a (randomized) differentially private one, *Fuzz* introduces a *probability monad* $\bigcirc$ and assigns monadic types to randomized computations. Accordingly, the concept of sensitivity is generalized so that the privacy cost of a differentially private function is interpreted as its sensitivity. Specifically, the type system guarantees that any program that can be given a type $!_\epsilon\mathbf{db} \multimap \bigcirc\mathbb{R}$ is differentially private, with privacy cost ϵ. [2]

The full *Fuzz* type system is described in [33, 34], which also show how several practical programs can be successfully type-checked; one of these is k-means clustering, a machine learning algorithm that takes a list of points and computes centers around which the points cluster. However, as mentioned above, a key limitation of *Fuzz* is that its sensitivity annotations are purely numeric. In the case of k-means clustering, this is a major inconvenience: the sensitivity depends on the number of iterations of the algorithm, but *Fuzz* types cannot express the fact that a function's sensitivity in one parameter may depend on the value of another—the only way out is to hard-code the number of iterations. For instance, in *Fuzz* a k-means program performing two iterations can be typed as:

$$\textit{2iter-k-means} : !_\infty\mathbf{L}(\mathbb{R}\otimes\mathbb{R}) \multimap !_{6\epsilon}\mathbf{Bag}(\mathbb{R}\otimes\mathbb{R}) \multimap \bigcirc(\mathbf{L}(\mathbb{R}\otimes\mathbb{R}))$$

This type says that *2iter-k-means* is a program that, when provided with an initial list of centers (of type $\mathbf{L}(\mathbb{R} \otimes \mathbb{R})$), produces a 6ϵ-differentially private algorithm that maps a dataset to a new list of centers. A k-means program that performs a different number of iterations would have a different type. Worse, in Section 6, we will see several examples of practical algorithms whose *types* cannot be expressed in *Fuzz*, e.g., the IDC algorithm [19], in which the desired sensitivity is itself a parameter. This means that we cannot even add such operations to *Fuzz* as primitives.

In *DFuzz*, we overcome this limitation by adding dependent types. In particular, we add a lightweight form of dependent types that allow us to give the following type to a general algorithm for k-means:

$$\textit{k-means} : \forall i, k.(!_\infty\mathbf{N}[i] \multimap !_\infty\mathbf{L}(\mathbb{R} \otimes \mathbb{R})[k] \multimap\ !_{3i\epsilon}\mathbf{Bag}(\mathbb{R} \otimes \mathbb{R}) \multimap \bigcirc(\mathbf{L}(\mathbb{R} \otimes \mathbb{R})[k]))$$

This type gives more precise information about the behavior of *k-means*: it says that *k-means* is a program that, when provided with a natural number i specifying the number of iterations (the sensitivity annotation is ∞ since this parameter does not need to be kept private) and a list of k centers, produces a $3i\epsilon$-differentially private algorithm that maps the dataset to a list of k centers. In the next section, we describe *DFuzz*'s dependent types in more detail.

3. DFuzz

The main novelty of *DFuzz* is the way it combines a lightweight form of dependent types, *à la* Dependent ML [38], with a sensitivity analysis inspired by the one in [33, 34]. This combination requires several ingredients, which are formally defined in Figure 1 (type grammar), Figure 2 (program syntax), Figure 3 (kinding rules), Figure 4 (subtyping rules), Figures 5, 6, 7 (typing rules), and Figure 8 (selected operational semantics rules). We describe each of these components below, omitting some secondary details for brevity. (In particular, we elide some linear data types like $\otimes$, $\&$, and $\oplus$. A full description is available in [14].)

[2] In [34], a *fixed* value of ϵ was baked into the metric on distributions. In contrast, we use the convention from [33], where the privacy parameter ϵ is not fixed, and is made explicit in the type annotation.

κ	::=	$\iota \mid \upsilon$	(kinds)
Z	::=	$\mathbf{N}[S] \mid \mathbf{L}(\tau)[S] \mid \mathbb{R}[R] \mid \mathcal{P}[S]$	(precise types)
α	::=	$\mathbb{R} \mid \mathtt{db}$	(basic types)
a	::=	$\forall i : \kappa.\tau \mid \exists i : \kappa.\tau$	(quantified types)
τ	::=	$a \mid Z \mid \alpha \mid A \multimap \tau \mid \bigcirc\tau$	(types)
A	::=	$!_R\tau$	(modal types)
S	::=	$i \mid 0 \mid S + 1$	(size terms)
R	::=	$k \mid r \mid S \mid R + R \mid R \cdot R \mid \infty$	(sensitivity terms)
Φ	::=	$\emptyset \mid S = 0 \mid S = i + 1 \mid \Phi \wedge \Phi$	(assumptions)
$\mathcal{C}$	::=	$S = S \mid R \leq R$	(constraints)
ϕ	::=	$\emptyset \mid \phi, i : \kappa$	(kind environments)
Γ	::=	$\emptyset \mid \Gamma, x : A$	(type environments)

Figure 1. *DFuzz* Types

p	::=	$(r_1, \ldots, r_n)$	(probabilities)
e, g	::=	$x \mid \lambda x.e \mid e\,e \mid \mathtt{f} \mid \mathbf{fix}\ x.e$ $\mid 0 \mid \mathbf{s}\,e \mid \mathtt{nil} \mid \mathtt{cons}[S](e, e)$ $\mid \mathbf{case_N}\ e\ \mathbf{of}\ 0 \to e \mid x[i] + 1 \to e$ $\mid \mathbf{case_L}\ e\ \mathbf{of}\ \mathtt{nil} \to e \mid \mathtt{cons}[i](y, x) \to e$ $\mid \Lambda i.e \mid e[I] \mid \mathbf{pack}(e, I)\ \mathbf{as}\ \exists i : \kappa.\sigma$ $\mid \mathbf{unpack}\ e\ \mathbf{as}\ (x, i)\ \mathbf{in}\ e$ $\mid p \mid \mathbf{return}\,e \mid \{e, (e_1, \ldots, e_n)\}$ $\mid \mathbf{let} \bigcirc x = e\ \mathbf{in}\ e$	(expressions)
v	::=	$\lambda x.e \mid \mathtt{f} \mid \mathbf{fix}\ x.e \mid ()$ $\mid 0 \mid \mathbf{s}\,v \mid \mathtt{nil} \mid \mathtt{cons}[S](v, v)$ $\mid \Lambda i.e \mid \mathbf{pack}(v, I)\ \mathbf{as}\ \exists i : \kappa.\sigma$ $\mid p \mid \mathbf{return}\,v \mid \{v, (e_1, \ldots, e_n)\}$	(values)
s	::=	$\mathbf{do}\,e \mid \{p, (s_1, \ldots, s_n)\}$	(states)
f	::=	$\mathbf{do\,return}\,v \mid \{p, (f_1, \ldots, f_n)\}$	(final states)

Figure 2. *DFuzz* Syntax

Sizes and sensitivities The special feature of *DFuzz* types (Figure 1) is that they can contain *size* or *sensitivity* information. This information is described by means of terms in a small language at the type level. A *size term*, written S, is a term that conforms to the grammar

$$S ::= i \mid 0 \mid S + 1$$

where i is a size variable. Size terms are simple expressions over natural numbers; they are used to describe the size of data types. A *sensitivity term*, denoted R, is a term that conforms to the grammar

$$R ::= k \mid r \mid S \mid R + R \mid R \cdot R \mid \infty$$

where k is a sensitivity variable and r is a non-negative real constant, i.e., $r \in \mathbb{R}^{\geq 0}$. Sensitivity terms are simple expressions that consist of positive real numbers and the symbol ∞; they are used to describe program sensitivity. An annotation of ∞ means that

$$\frac{}{\phi \vdash 0 : \iota}\ (k.0) \qquad \frac{}{\phi \vdash \infty : \upsilon}\ (k.\infty) \qquad \frac{\phi(i) = \kappa}{\phi \vdash i : \kappa}\ (k.Ax) \qquad \frac{r \in \mathbb{R}^{\geq 0}}{\phi \vdash r : \upsilon}\ (k.\mathbb{R}) \qquad \frac{\phi \vdash S : \iota}{\phi \vdash S + 1 : \iota}\ (k.+1)$$

$$\frac{\phi \vdash R_1 : \upsilon \quad \phi \vdash R_2 : \upsilon}{\phi \vdash R_1 + R_2 : \upsilon}\ (k.+) \qquad \frac{\phi \vdash R_1 : \upsilon \quad \phi \vdash R_2 : \upsilon}{\phi \vdash R_1 \cdot R_2 : \upsilon}\ (k.\cdot) \qquad \frac{\phi \vdash S : \iota}{\phi \vdash S : \upsilon}\ (k.<:)$$

Figure 3. Kinding rules

$$\frac{}{\phi; \Phi \models \sigma \sqsubseteq \sigma}\ (st.R) \qquad \frac{\phi; \Phi \models \sigma_1 \sqsubseteq \sigma_2 \quad \phi; \Phi \models \sigma_2 \sqsubseteq \sigma_3}{\phi; \Phi \models \sigma_1 \sqsubseteq \sigma_3}\ (st.T) \qquad \frac{\phi; \Phi \models \sigma_3 \sqsubseteq \sigma_1 \quad \phi; \Phi \models \sigma_2 \sqsubseteq \sigma_4}{\phi; \Phi \models \sigma_1 \multimap \sigma_2 \sqsubseteq \sigma_3 \multimap \sigma_4}\ (st.\multimap)$$

$$\frac{\phi, i : \kappa; \Phi \models \sigma_1 \sqsubseteq \sigma_2 \quad i \notin \mathrm{FSV}(\Phi)}{\phi; \Phi \models \forall i : \kappa.\sigma_1 \sqsubseteq \forall i : \kappa.\sigma_2}\ (st.\forall) \qquad \frac{\phi; \Phi \models R_2 \leq R_1 \quad \phi; \Phi \models \sigma_1 \sqsubseteq \sigma_2}{\phi; \Phi \models !_{R_1}\sigma_1 \sqsubseteq !_{R_2}\sigma_2}\ (st.!)$$

$$\frac{\phi, i : \kappa; \Phi \models \sigma_1 \sqsubseteq \sigma_2 \quad i \notin \mathrm{FSV}(\Phi)}{\phi; \Phi \models \exists i : \kappa.\sigma_1 \sqsubseteq \exists i : \kappa.\sigma_2}\ (st.\exists) \qquad \frac{\phi; \Phi \models S = S'}{\phi; \Phi \models \mathbf{N}[S] \sqsubseteq \mathbf{N}[S']}\ (st.\mathbf{N}) \qquad \frac{\phi; \Phi \models S = S' \quad \phi; \Phi \models \sigma \sqsubseteq \tau}{\phi; \Phi \models \mathbf{L}(\sigma)[S] \sqsubseteq \mathbf{L}(\tau)[S']}\ (st.\mathbf{L}(\sigma))$$

Figure 4. Subtyping rules

$$\frac{}{\phi; \Phi; \Gamma, x : !_1\sigma \vdash x : \sigma}\ (Ax) \qquad \frac{\phi; \Phi; \Gamma \vdash e : \sigma \quad \phi; \Phi \models \sigma \sqsubseteq \tau}{\phi; \Phi; \Gamma \vdash e : \tau}\ (\sqsubseteq.R) \qquad \frac{\phi; \Phi; \Gamma \vdash e : \sigma \quad \phi; \Phi \models \Delta \sqsubseteq \Gamma}{\phi; \Phi; \Delta \vdash e : \sigma}\ (\sqsubseteq.L)$$

$$\frac{\phi; \Phi; \Gamma, x : !_\infty\sigma \vdash e : \sigma}{\phi; \Phi; \infty \cdot \Gamma \vdash \mathbf{fix}\ x.e : \sigma}\ (\mathit{fix}) \qquad \frac{\phi; \Phi; \Gamma, x : !_R\sigma \vdash e : \tau}{\phi; \Phi; \Gamma \vdash \lambda x.e : !_R\sigma \multimap \tau}\ (\multimap I) \qquad \frac{\phi; \Phi; \Gamma \vdash e : !_R\sigma \multimap \tau \quad \phi; \Phi; \Delta \vdash g : \sigma}{\phi; \Phi; \Gamma + R \cdot \Delta \vdash e\ g : \tau}\ (\multimap E)$$

Figure 5. Core Typing Rules

$$\frac{\phi; \Phi; \Gamma \vdash e : \mathbf{N}[S]}{\phi; \Phi; \Gamma \vdash \mathbf{s}\, e : \mathbf{N}[S+1]}\ (\mathbf{s}) \qquad \frac{}{\phi; \Phi; \Gamma \vdash \mathtt{nil} : \mathbf{L}(\sigma)[0]}\ (n) \qquad \frac{\phi; \Phi; \Gamma \vdash e : \sigma \quad \phi; \Phi; \Delta \vdash g : \mathbf{L}(\sigma)[S]}{\phi; \Phi; \Gamma + \Delta \vdash \mathtt{cons}[S](e, g) : \mathbf{L}(\sigma)[S+1]}\ (c)$$

$$\frac{}{\phi; \Phi; \Gamma \vdash 0 : \mathbf{N}[0]}\ (0) \qquad \frac{\phi; \Phi; \Gamma \vdash e : \mathbf{N}[S] \quad \phi; \Phi \wedge S = 0; \Delta \vdash g_1 : \sigma \quad \phi, i : \iota; \Phi \wedge S = i + 1; \Delta, x : !_R\mathbf{N}[i] \vdash g_2 : \sigma}{\phi; \Phi; \Delta + R \cdot \Gamma \vdash \mathbf{case}_{\mathbf{N}}\, e\, \mathbf{of}\, 0 \to g_1 \mid x[i] + 1 \to g_2 : \sigma}\ (\mathbf{case})_{\mathbf{N}}$$

$$\frac{\phi; \Phi; \Gamma \vdash e : \mathbf{L}(\tau)[S] \quad \phi; \Phi \wedge S = 0; \Delta \vdash g_1 : \sigma \quad \phi, i : \iota; \Phi \wedge S = i + 1; \Delta, x : !_R\mathbf{L}(\tau)[i], y : !_R\tau \vdash g_2 : \sigma}{\phi; \Phi; \Delta + R \cdot \Gamma \vdash \mathbf{case}_{\mathbf{L}}\, e\, \mathbf{of}\, \mathtt{nil} \to g_1 \mid \mathtt{cons}(y, x[i]) \to g_2 : \sigma}\ (\mathbf{case})_{\mathbf{L}(\tau)}$$

$$\frac{\phi, i : \kappa; \Phi; \Gamma \vdash e : \tau \quad i \notin \mathrm{FSV}(\Phi; \Gamma)}{\phi; \Phi; \Gamma \vdash \Lambda i.e : \forall i : \kappa.\tau}\ (\forall I) \qquad \frac{\phi; \Phi; \Gamma \vdash e : \tau\{I/i\} \quad \phi \vdash I : \kappa}{\phi; \Phi; \Gamma \vdash \mathbf{pack}(e, I)\ \mathbf{as}\ \exists i : \kappa.\tau : \exists i : \kappa.\tau}\ (\exists I)$$

$$\frac{\phi; \Phi; \Gamma \vdash e : \forall i : \kappa.\tau \quad \phi \vdash I : \kappa}{\phi; \Phi; \Gamma \vdash e[I] : \tau\{I/i\}}\ (\forall E) \qquad \frac{\phi; \Phi; \Gamma \vdash e : \exists i : \kappa.\tau \quad \phi, i : \kappa; \Phi; \Delta, x : !_R\tau \vdash g : \sigma \quad i \notin \mathrm{FSV}(\Phi; \Delta; \sigma; R)}{\phi; \Phi; \Delta + R \cdot \Gamma \vdash \mathbf{unpack}\ e\ \mathbf{as}\ (x, i)\ \mathbf{in}\ g : \sigma}\ (\exists E)$$

$$\frac{\phi; \Phi \text{ imply } f\ R\text{-sensitive } \sigma \to \tau}{\phi; \Phi; \Gamma \vdash \mathtt{f} : !_R\sigma \multimap \tau}\ (\mathit{Ext})$$

Figure 6. Data Types and Polymorphism Typing Rules

$$\frac{\phi; \Phi; \Gamma \vdash e : \sigma}{\phi; \Phi; \infty\Gamma \vdash \mathbf{return}\ e : \bigcirc\sigma}\ (\bigcirc I) \qquad \frac{\phi; \Phi; \Gamma \vdash e : \bigcirc\sigma \quad \phi; \Phi; \Delta, x : !_\infty\sigma \vdash g : \bigcirc\tau}{\phi; \Phi; \Gamma + \Delta \vdash \mathbf{let}\, \bigcirc x = e\, \mathbf{in}\, g : \bigcirc\tau}\ (\bigcirc E) \qquad \frac{}{\phi; \Phi; \Gamma \vdash (r_1, \ldots, r_n) : \mathcal{P}[n]}\ (\mathcal{P})$$

$$\frac{\phi; \Phi; \Gamma \vdash e : \mathcal{P}[n] \quad \phi; \Phi; \Delta \vdash e_i : \bigcirc\sigma \quad (\forall i)}{\phi; \Phi; \Gamma + \Delta \vdash \{e, (e_1, \ldots, e_n)\} : \bigcirc\sigma}\ (\{e\}) \qquad \frac{\phi; \Phi; \Gamma \vdash p : \mathcal{P}[n] \quad \phi; \Phi; \Delta \vdash s_i : \sigma \quad (\forall i)}{\phi; \Phi; \Gamma + \Delta \vdash \{p, (s_1, \ldots, s_n)\} : \sigma}\ (\{s\}) \qquad \frac{\phi; \Phi; \Gamma \vdash e : \bigcirc\sigma}{\phi; \Phi; \Gamma \vdash \mathbf{do}\ e : \sigma}\ (\mathbf{do})$$

Figure 7. Probability Layer Typing Rules

there is no guaranteed bound on the sensitivity. The binary operators $+$ and $\cdot$ are the symbolic counterparts on sensitivities of sum and product on values. Their precise meaning is described below. We will use the metavariable I to range over sizes and sensitivities.

Any size term is also a sensitivity term. This is important for expressing *dependencies* between sizes (for example, number of iterations) and sensitivities (for example, privacy cost). To ensure the correct behavior of size and sensitivity terms, and to prevent undesired substitutions, we consider size and sensitivity terms to be typed as well; to avoid confusion, we refer to the types for size and sensitivity terms as *kinds*. *DFuzz* uses two kinds: ι for size annotations and υ for sensitivity annotations. Kinds are assigned to terms via *kind judgments* of the form

$$\phi \vdash I : \kappa$$

where ϕ is a *kind environment*, i.e., a set of kind assignments to size and sensitivity variables. The rules for deriving the judgment $\phi \vdash I : \kappa$ are presented in Figure 3. Notice that we have a *subkind* relation on terms that is induced by the rule $(k.<:)$. This relation allows us to consider size terms as sensitivity terms. For notational convenience, we will sometime write sizes and sensitivities terms without making explicit their kind.

We can interpret size terms over the domain $\mathbb{N}$ of natural numbers and sensitivity terms over the domain $\mathbb{R}^{\geq 0} \cup \{\infty\}$ of non-negative extended real numbers. To interpret expressions with free variables we need, as usual, a notion of *assignment*—that is, a mapping ρ from size variables to elements of $\mathbb{N}$ and from sensitivity variables to elements of $\mathbb{R}^{\geq 0} \cup \{\infty\}$. We write $\mathrm{dom}(\rho)$ for the domain of ρ; this is a set of variable-kind mappings of the shape $i : \kappa$ where each variable i appears at most once.

Given a term $\phi \vdash I : \kappa$ and an assignment ρ such that $\phi \subseteq \mathrm{dom}(\rho)$, we inductively define the interpretation $[\![I]\!]_\rho$:

- $[\![0]\!]_\rho = 0$
- $[\![i]\!]_\rho = \rho(i)$
- $[\![S+1]\!]_\rho = [\![S]\!]_\rho + 1$
- $[\![r]\!]_\rho = r$
- $[\![R_1 + R_2]\!]_\rho = [\![R_1]\!]_\rho \mathbin{\hat{+}} [\![R_2]\!]_\rho$
- $[\![R_1 \cdot R_2]\!]_\rho = [\![R_1]\!]_\rho \mathbin{\hat{\cdot}} [\![R_2]\!]_\rho$
- $[\![\infty]\!]_\rho = \infty$

where $\hat{+}$ and $\hat{\cdot}$ are the usual sum and product extended to ∞:

$$\begin{aligned} &\infty \mathbin{\hat{+}} r = r \mathbin{\hat{+}} \infty = \infty && \text{for every } r \in \mathbb{R}^{\geq 0} \cup \{\infty\}, \\ &\infty \mathbin{\hat{\cdot}} r = r \mathbin{\hat{\cdot}} \infty = \infty && \text{for every } r \neq 0, \text{ and} \\ &\infty \mathbin{\hat{\cdot}} 0 = 0 \mathbin{\hat{\cdot}} \infty = 0 \end{aligned}$$

(Note that this definition implicitly coerces size terms—natural numbers—into real numbers as needed.) The well-definedness of the interpretation is ensured by the following lemma:

3.1 Lemma: Let $\phi \vdash I : \kappa$ and $\phi \subseteq \mathrm{dom}(\rho)$. Then:

- If $\kappa = \iota$, then $[\![I]\!]_\rho \in \mathbb{N}$.
- If $\kappa = \upsilon$, then $[\![I]\!]_\rho \in \mathbb{R}^{\geq 0} \cup \{\infty\}$.

Proof: By induction on the derivation proving $\phi \vdash S : \kappa$. □

Data types and dependent pattern matching Our syntax of types offers just two representative algebraic data types: natural numbers $\mathbf{N}[S]$ and lists $\mathbf{L}(\sigma)[S]$ with elements of type σ (see Figure 1). We could have gone further and included general algebraic datatype declarations and pattern matching, but this would add to the complexity of the notation without (we believe) raising new conceptual issues; we leave this generalization to future work.

Intuitively, the size of a natural number corresponds to its value (we consider a unary encoding of the natural numbers, and we assume the natural number 0 to be of size 0), and the size of a list corresponds to the number of elements in it (we assume `nil` to be of size 0). Types like $\mathbf{N}[S]$ and $\mathbf{L}(\sigma)[S]$ are inhabited only by terms whose evaluation produces a value of size S (if it terminates). The approach is reminiscent of Hayashi's singleton types [21].

To illustrate why size terms are helpful for expressiveness, let us consider the list data type in more detail. Values of the list type are built through the constructors `nil` and $\texttt{cons}[S](e, g)$ (see Figure 2), where `cons` carries the size S of the expression g as additional information. This information is used to support a simple form of *dependent pattern matching*: the $\mathbf{case_L}$ destructor can, in addition to the usual term-level pattern matching, also perform type-level pattern matching. This is described by the following reduction rule (see Figure 8)

$$\begin{aligned} \mathbf{case_L}\ \texttt{cons}[S](e_1, e_2)\ \mathbf{of}\ \texttt{nil} \to g_1 \mid \texttt{cons}[i](y, x) \to g_2 \\ \mapsto g_2\{e_1/y\}\{e_2/x\}\{S/i\} \end{aligned}$$

in which the size S is propagated to the term g_2. In a similar way, in the reduction rule ($\text{Op-Case}_\mathbf{s}$) in Figure 8 for dependent pattern matching on $\mathbf{N}[S]$, the information n is propagated to the term g_2.

The data type elimination rules in Figure 6 are crucial to making this technique work. For instance, the elimination rule for the list data type

$$\frac{\begin{array}{c} \phi; \Phi; \Gamma \vdash e : \mathbf{L}(\tau)[S] \qquad \phi; \Phi \wedge S = 0; \Delta \vdash g_1 : \sigma \\ \phi, i : \iota; \Phi \wedge S = i + 1; \Delta, x : !_R \mathbf{L}(\tau)[i], y : !_R \tau \vdash g_2 : \sigma \end{array}}{\phi; \Phi; \Delta + R \cdot \Gamma \vdash \mathbf{case_L}\ e\ \mathbf{of}\ \texttt{nil} \to g_1 \mid \texttt{cons}[i](y, x) \to g_2 : \sigma}$$

adds assumptions (explained in detail below) to the contexts used to typecheck the two branches of the **case** construct, reflecting information gleaned by the **case** about the size S of the test value e. In the first branch, it is assumed that $S = 0$; this extra piece of information enables us to remember, at the typing judgment level, that e has been matched to `nil`. Conversely, in the second branch, it is assumed that $S = i + 1$, giving us a name i for the size of the tail of the list e, which appears in the type assumed for the pattern variable x. A similar mechanism is used in the elimination rule for the $\mathbf{N}[S]$ data type.

Assumptions and constraints More formally, a judgment

$$\phi; \Phi; \Gamma \vdash e : \sigma$$

contains, besides the usual environment Γ for term variables and the kind environment ϕ, an extra parameter Φ that records the assumptions under which the typing is obtained. Intuitively, the expression e can be assigned the type σ for any value of its free size variables satisfying Φ. Since assumptions Φ record the size constraints generated by the pattern matching rules, we consider assumptions that conform to the following grammar:

$$\Phi ::= \emptyset \mid S = 0 \mid S = i + 1 \mid \Phi \wedge \Phi$$

An assumption Φ is well defined when it is associated with a kind environment ϕ that determines the kinds of the free size variables in Φ. We write $\phi \vdash \Phi$ in this case. Given an assumption $\phi \vdash \Phi$ and an assignment ρ such that $\phi \subseteq \mathrm{dom}(\rho)$, the interpretation $[\![\Phi]\!]_\rho$ is defined inductively as follows:

- $[\![\emptyset]\!]_\rho = \mathbf{true}$
- $[\![S = 0]\!]_\rho = ([\![S]\!]_\rho \mathbin{\hat{=}} 0)$
- $[\![S = i + 1]\!]_\rho = ([\![S]\!]_\rho \mathbin{\hat{=}} [\![i]\!]_\rho \mathbin{\hat{+}} 1)$
- $[\![\Phi_1 \wedge \Phi_2]\!]_\rho = [\![\Phi_1]\!]_\rho \wedge [\![\Phi_2]\!]_\rho$

where $\hat{=}$ is equality on natural numbers. The use of assumptions is crucial for the subtyping relation (Figure 4). Indeed, subtyping

judgments involve judgments of the shape

$$\phi; \Phi \models \mathcal{C}$$

where $\mathcal{C}$ is a *constraint* of the form $R \leq R'$ or $S = S'$. The meaning of a constraint $\mathcal{C}$ also follows directly from the interpretation of the size and sensitivity terms it contains. Given an assignment ρ for the size and sensitivity variables appearing in $\mathcal{C}$, the interpretation $[\![\mathcal{C}]\!]_\rho$ is defined inductively as

- $[\![R \leq R']\!]_\rho = ([\![R]\!]_\rho \mathrel{\hat{\leq}} [\![R']\!]_\rho)$
- $[\![S = S']\!]_\rho = ([\![S]\!]_\rho \mathrel{\hat{=}} [\![S']\!]_\rho)$

where $\hat{\leq}$ is the expected extension of $\leq$ to cover ∞. The judgment $\phi; \Phi \models \mathcal{C}$ asserts that the constraint $\mathcal{C}$ is a logical consequences of the assumption Φ, i.e. for every ρ such that $[\![\Phi]\!]_\rho = \mathbf{true}$ we have $[\![\mathcal{C}]\!]_\rho = \mathbf{true}$. Analogously, we will also use the notation $\phi; \Phi \models \Psi$ to denote the fact that the assumption Ψ is a logical consequence of the assumption Φ. We will see how these judgments are used later when we present the subtyping rules in detail.

Generalized sensitivities and the scaling modality Sensitivity terms are the key ingredients in the sensitivity analysis. They appear as decorations of the *scaling modality* $!_R$. The scaling modality is used to define types with the shape $!_R\sigma$, which are used in turn in function types and in term environments. Suppose that a given expression e can be assigned the function type $!_R\sigma \multimap \tau$:

$$\phi; \Phi; \Gamma \vdash e \ : \ !_R\sigma \multimap \tau$$

Then we know that e is a function that maps elements of type σ to elements of type τ as long as Φ is satisfied, and that in this case R is an upper bound on the function's sensitivity.

Two points merit further explanation. First, a static analysis cannot always describe the exact sensitivity of a program; we are always dealing with conservative upper bounds. Second, we say that R "describes" this upper bound because it is not necessarily a number: it can depend on size or sensitivity variables from ϕ, which can themselves be involved in assumptions from Φ. This dependency is central to the expressivity of *DFuzz*.

Modal types also appear in type environments. A typing judgment of the form

$$\phi; \Phi; x_1 : !_{R_1}\sigma_1, \ldots, x_n : !_{R_n}\sigma_n \vdash e : \sigma$$

says that the expression e is a function whose sensitivity in the i^{th} argument is described by the sensitivity term R_i. Notice that this means that every function type comes with a sensitivity term associated to its input type. Indeed, here—unlike [33, 34]—we are implicitly using the standard linear logic decomposition of the intuitionistic implication $\sigma \to \tau = !\sigma \multimap \tau$ and the observation that it suffices to have modal types appearing in negative position. For convenience, we will similarly use the notation $\sigma \to \tau$ as a shorthand for non-sensitive functions, i.e., functions of a type $!_R\sigma \multimap \tau$ where $[\![R]\!] = \infty$.

Modal types can be combined by arithmetic operations

$$!_R\sigma + !_T\sigma = !_{R+T}\sigma \ \text{ and } \ T \cdot !_R\sigma = !_{T \cdot R}\sigma$$

and these operations are lifted to type environments. The sum of two contexts is defined inductively as:

- $\emptyset + \emptyset = \emptyset$
- $(\Gamma, x : A) + (\Delta, x : B) = (\Gamma + \Delta), x : A + B$
- $(\Gamma, x : A) + \Delta = \Gamma + (\Delta, x : A) = (\Gamma + \Delta), x : A$, if $x \notin \mathrm{dom}(\Gamma, \Delta)$

The product of a type environment with a sensitivity term T is:

- $T \cdot \emptyset = \emptyset$
- $T \cdot (\Gamma, x : A) = T \cdot \Gamma, x : T \cdot A$

It is worth emphasizing that the sum $\Gamma + \Delta$ of two type environments Γ and Δ is defined only in the case that for each $x \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta)$ the type $A + B$ is defined, where $x : A \in \Gamma$ and $x : B \in \Delta$. This in turn requires that there is a type σ such that $A = !_{R_1}\sigma$ and $B = !_{R_2}\sigma$. In all the binary rules in Figure 5 we implicitly assume this condition.

A sensitivity term can also appear as the annotation of a precise type $\mathbb{R}[R]$. This class of types is close in spirit to the data types $\mathbf{N}[S]$ and $\mathbf{L}(\sigma)[S]$, since it classifies terms whose value is described by R. However, we have no destructor operation for this kind of type. We will see in Section 5.4 that these types can be used to dynamically specify the sensitivity of certain operations.

Subtyping and quantifiers The assumptions introduced by pattern matching play a crucial role in the subtyping relation $\sqsubseteq$, which is defined by the rules in Figure 4. A subtyping judgment has the shape

$$\phi; \Phi \vdash \sigma \sqsubseteq \tau$$

where ϕ is a kind environment and Φ is an assumption. Intuitively, the subtyping relation $\phi; \Phi \vdash \sigma \sqsubseteq \tau$ says that (1) σ and τ are equal up to their decorations with size and sensitivity terms, and (2) the decoration of τ is more permissive than the decoration of σ under the assumption Φ. One main purpose of the subtyping relation is to capture the fact that an R-sensitive function is also R'-sensitive for $R \leq R'$. This is ensured by the rule for the scaling modality:

$$\frac{\phi; \Phi \models R_2 \leq R_1 \quad \phi; \Phi \models \sigma_1 \sqsubseteq \sigma_2}{\phi; \Phi \models !_{R_1}\sigma_1 \sqsubseteq !_{R_2}\sigma_2} \ (st.!)$$

Indeed, the combination of this rule and the rule for function types

$$\frac{\phi; \Phi \models \sigma_3 \sqsubseteq \sigma_1 \quad \phi; \Phi \models \sigma_2 \sqsubseteq \sigma_4}{\phi; \Phi \models \sigma_1 \multimap \sigma_2 \sqsubseteq \sigma_3 \multimap \sigma_4} \ (st.\multimap)$$

(which, as usual, is contravariant in its first argument) ensures that $\phi; \Phi \vdash !_R\sigma \multimap \tau \sqsubseteq !_{R'}\sigma \multimap \tau$ iff $\phi; \Phi \models R \leq R'$. In other words, if we can prove that $R \leq R'$ under assumption Φ, then every expression that can be given the type $!_R\sigma \multimap \tau$ can *also* be given the type $!_{R'}\sigma \multimap \tau$.

Subtyping is also needed for putting the dependent pattern matching mechanism to work, to unify the contexts and the types in the two branches of a $\mathbf{case}$ rule. This is obtained by using subtyping judgments on data types. For instance, for natural numbers we have a rule

$$\frac{\phi; \Phi \models S = S'}{\phi; \Phi \models \mathbf{N}[S] \sqsubseteq \mathbf{N}[S']} \ (st.\mathbf{N})$$

that allows us to obtain the type $\mathbf{N}[S']$ from the type $\mathbf{N}[S]$ if we have $\phi; \Phi \models S = S'$.

Subtyping can be applied by using the rule $(\sqsubseteq.R)$ in Figure 5. Moreover, it can be extended to type environments: the judgment $\phi; \Phi \models \Gamma \sqsubseteq \Delta$ holds if for every $x : A \in \Delta$ we have $x : B \in \Gamma$ and $\phi; \Phi \models B \sqsubseteq A$. Note that the type environment Γ can contain variable assignments for variables that do not appear in Δ. Thus the rule $(\sqsubseteq.L)$ in Figure 5 can also be seen as a kind of weakening.

In order to be able to express the size and sensitivity dependencies in a uniform way, , we additionally enrich the type language with a universal quantifier $\forall$ and an existential quantifier $\exists$ over size and sensitivity variables (Figure 6). For instance, the type

$$\forall i : \iota.(\mathbf{N}[i] \multimap !_i\mathbb{R} \multimap \mathbb{R})$$

can be assigned to a program that, given a value of type $\mathbf{N}[S]$, returns an S-sensitive function of type $!_S\mathbb{R} \multimap \mathbb{R}$, for any size term $\vdash S : \iota$. Similarly, the type

$$\forall i : \iota.(\mathbf{N}[i] \multimap \exists k : \upsilon.(\ !_j\mathbb{R} \multimap \mathbb{R}))$$

abstracts the sensitivity of the function that is returned when a term of this type is provided with a value of type $\mathbf{N}[S]$. Finally, the type

$$\exists i : \iota.\mathbf{N}[i]$$

is the type of the ordinary natural numbers.

The probability layer To ensure differential privacy, we need to be able to write probabilistic computations. This is achieved in *DFuzz* by considering a grammar of programs with two layers (Figure 2) . We have an *expression layer* that contains the constructors and destructors for each data type, and an additional *probability state layer* that contains the program constructions for describing probabilities over expressions and values. To mediate between the two layers, we use a monadic type $\bigcirc\tau$, similar to the one found in Ramsey and Pfeffer's stochastic lambda calculus [32].

In order to describe probabilistic states, we need to have probability vectors p, i.e., lists of real numbers from the interval $[0, 1]$ that sum up to 1. Probability vectors can be typed by means of a precise type of the shape $\mathcal{P}[S]$, where the size term S describes the length of the vector. Though S can range over variables, the typing rules for vectors only use types $\mathcal{P}[n]$ indexed by a constant natural number n. A *probabilistic state* s then is either an object of the shape $\{p, (s_1, \ldots, s_n)\}$ where p is a probabilistic vector, or an object of the shape $\mathbf{do}\, e$. Intuitively, the former associates a discrete probability distribution, described by p, to a list of probabilistic states $(s_1, \ldots, s_n)$, whereas the latter turns an expression into a probabilistic state. Probabilistic states represent probabilistic computations in the sense that their evaluation results in *final states* that assign probabilities to values. For an example, the state

$$\{(0.2, 0.8), (\mathbf{do}\,\mathbf{return}\,1, \{(0.5, 0.5), (\mathbf{do}\,\mathbf{return}\,2, \mathbf{do}\,\mathbf{return}\,3)\})\}$$

is a final state in which the value 1 is returned with probability 0.2 and the values 2 and 3 are each returned with probability $0.8 \cdot 0.5 = 0.4$.

We use three monadic expression forms to ensure that only expressions representing probabilistic computations can be turned into probabilistic states. The expression $\mathbf{return}\, e$ can be seen as representing the distribution that deterministically yields e; the monadic sequencing $\mathbf{let} \bigcirc x = e\, \mathbf{in}\, e'$ can be seen as a program that draws a sample x from the probabilistic computation e and then continues with the computation e'; and the explicit probability construction $\{e, (e_1, \ldots, e_n)\}$ associates an expression e representing a probability vector with a list of random computations $(e_1, \ldots, e_n)$. The typing rules (Figure 7) ensure that we consider only well-formed expressions. In particular, the rule

$$\frac{\phi; \Phi; \Gamma \vdash e : \mathcal{P}[n] \quad \phi; \Phi; \Delta \vdash e_i : \bigcirc\sigma \quad (\forall i)}{\phi; \Phi; \Gamma + \Delta \vdash \{e, (e_1, \ldots, e_n)\} : \bigcirc\sigma}\ (\{e\})$$

ensures that (1) e represents a probability vector, and (2) every e_i is a probabilistic computation. Notice also that this rule is responsible for the well-formedness of the probabilistic states. It ensures that we associate lists of expressions of length n only with probability vectors of the same length. This is the reason for introducing the precise type $\mathcal{P}[n]$.

As a last remark, notice that the different components of a state represent independent computations, so, even though evaluation of expressions is defined sequentially, we define evaluation on states in parallel.

Sensitivity and primitive operations One last component that is necessary to make the framework practical is a way to add *trusted* primitive operations, i.e., operations that are known to preserve the properties of the type system. This is intuitively the meaning of the following typing rule from Figure 6:

$$\frac{\phi; \Phi \text{ imply } f\ R\text{-sensitive } \sigma \to \tau}{\phi; \Phi; \Gamma \vdash \mathtt{f} : !_R\sigma \multimap \tau}\ (Ext)$$

which says that we can add to *DFuzz* any primitive operation $\mathtt{f}$ of type $!_R\sigma \multimap \tau$ as long as we know that this operation represents a mathematical function f that maps values in the type σ to values in the type τ and that is R-sensitive under the assumption Φ.

At the operational level, the evaluation rules for the primitive operation $\mathtt{f}$ must respect the behavior of the function f. This is obtained by means of the two following rules (Figure 8):

$$\frac{e \mapsto e'}{\mathtt{f}\, e \mapsto \mathtt{f}\, e'}\ \text{(Op-Ext-Tr)} \qquad \frac{}{\mathtt{f}\, v \mapsto f(v)}\ \text{(Op-Ext)}$$

We can also extend *DFuzz* with additional primitive types, as long as they are equipped with a metric that respects the properties we describe in Section 5.

4. Basic Metatheory

In this section, we develop fundamental properties of *DFuzz*. In order to show the usual properties one would expect from our programming language—type preservation and progress—we additionally need to prove some properties that are particular to our use of size and sensitivity annotations. These will also be useful to prove the Metric Preservation Theorem in the next section. We give just proof sketches; full proofs are in [14].

4.1 Properties of Sizes and Sensitivities

As we saw earlier, typing judgments in *DFuzz* have the form

$$\phi; \Phi; \Gamma \vdash e : \sigma$$

i.e., they are indexed by a set of kind assignments to index variables ϕ and an assumption Φ. Intuitively, the fact that the subtyping can prove statements using the assumptions in Φ is what makes the dependent pattern matching work: it enables us to recover the same type from the different branches.

Here, we prove some properties of the typing with respect to the assumption Φ. The first says that strengthening the assumption preserves the typing.

4.1.1 Lemma [Assumption Strengthening]: Suppose that $\phi; \Psi \models \Phi$. Then, we have:

1. If $\phi; \Phi \models \mathcal{C}$, then $\phi; \Psi \models \mathcal{C}$.
2. If $\phi; \Phi \models \sigma \sqsubseteq \tau$, then $\phi; \Psi \models \sigma \sqsubseteq \tau$.
3. If $\phi; \Phi; \Gamma \vdash e : \sigma$, then $\phi; \Psi; \Gamma \vdash e : \sigma$.
4. If $\phi; \Phi; \Gamma \vdash s : \sigma$, then $\phi; \Psi; \Gamma \vdash s : \sigma$.

Proof: (1) follows directly from the transitivity of the logical implication; (2) follows by induction on the derivation proving $\phi; \Phi \models \sigma \sqsubseteq \tau$, using Point 1 when needed. (3) follows by induction on the derivation proving $\phi; \Phi; \Gamma \vdash e : \sigma$, using Point 2 when needed. (4) follows by induction on the derivation proving $\phi; \Phi; \Gamma \vdash s : \sigma$, using Point 3 when needed. □

The environment ϕ can contain free size and sensitivity variables; these can be thought of as placeholders for any size or sensitivity index term, and they can be instantiated with a concrete index term when necessary. This is captured by the next lemma:

4.1.2 Lemma [Instantiation]:

1. If $\phi, i : \kappa; \Phi \models \mathcal{C}$, then, for every $\phi \vdash I : \kappa$, we have $\phi; \Phi\{I/i\} \models \mathcal{C}\{I/i\}$.
2. If $\phi, i : \kappa; \Phi \models \sigma \sqsubseteq \tau$, then, for every $\phi \vdash I : \kappa$, we have $\phi; \Phi\{I/i\} \models \sigma\{I/i\} \sqsubseteq \tau\{I/i\}$.

$$\frac{}{(\mathbf{case_N}\ 0\ \mathbf{of}\ 0 \to e_1 \mid x[i]+1 \to e_2) \mapsto e_1}\ \text{(Op-Case}_0\text{)} \qquad \frac{}{(\mathbf{case_N}\ \mathbf{s}\ n\ \mathbf{of}\ 0 \to e_1 \mid x[i] \to e_2) \mapsto e_2\{n/x\}\{n/i\}}\ \text{(Op-Case}_\mathbf{s}\text{)}$$

$$\frac{e \mapsto e'}{(\mathbf{case_L}\ e\ \mathbf{of}\ \mathtt{nil} \to e_1 \mid \mathtt{cons}[i](x,y) \to e_2) \mapsto (\mathbf{case_L}\ e'\ \mathbf{of}\ \mathtt{nil} \to e_1 \mid \mathtt{cons}[i](x,y) \to e_2)}\ \text{(Op-Case-List-Tr)}$$

$$\frac{e \mapsto e'}{\mathtt{f}\ e \mapsto \mathtt{f}\ e'}\ \text{(Op-Ext-Tr)} \qquad \frac{}{(\mathbf{case_L}\ \mathtt{nil}\ \mathbf{of}\ \mathtt{nil} \to e_1 \mid \mathtt{cons}(x, y[i]) \to e_2) \mapsto e_1}\ \text{(Op-Case}_{\mathtt{nil}}\text{)} \qquad \frac{}{\mathtt{f}\ v \mapsto f(v)}\ \text{(Op-Ext)}$$

$$\frac{}{(\mathbf{case_L}\ \mathtt{cons}[S](v,w)\ \mathbf{of}\ e_1 \mid \mathtt{cons}(x, y[i]).e_2) \mapsto e_2\{v/x\}\{w/y\}\{S/i\}}\ \text{(Op-Case}_\mathbf{c}\text{)} \qquad \frac{}{(\mathbf{fix}\ x.e)v \mapsto e\{\mathbf{fix}\ x.e/x\}v}\ \text{(Op-fix}_{\multimap}\text{)}$$

$$\frac{}{(\mathbf{fix}\ x.e)[I] \mapsto e\{\mathbf{fix}\ x.e/x\}[I]}\ \text{(Op-fix}_{\forall}\text{)} \qquad \frac{e \mapsto e'}{\mathbf{unpack}\ e\ \mathbf{as}\ (x,i)\ \mathbf{in}\ g \mapsto \mathbf{unpack}\ e'\ \mathbf{as}\ (x,i)\ \mathbf{in}\ g}\ \text{(Op-}\exists\text{-Tr-1)}$$

$$\frac{e \mapsto e'}{\mathbf{pack}(e, I)\ \mathbf{as}\ \exists i : \kappa.\sigma \mapsto \mathbf{pack}(e', I)\ \mathbf{as}\ \exists i : \kappa.\sigma}\ \text{(Op-}\exists\text{-Tr-2)}$$

$$\frac{}{\mathbf{unpack}\ (\mathbf{pack}(v, I)\ \mathbf{as}\ \exists i : \kappa.\sigma)\ \mathbf{as}\ (x,i)\ \mathbf{in}\ e \mapsto e\{I/i\}\{v/x\}}\ \text{(Op-}\exists\text{)}$$

$$\frac{e \mapsto e'}{\mathbf{return}\ e \mapsto \mathbf{return}\ e'}\ \text{(Op-Return-Tr)} \qquad \frac{e \mapsto e'}{\{e, (e_1, \ldots, e_n)\} \mapsto \{e', (e_1, \ldots, e_n)\}}\ \text{(Op-P-Exp-Tr)}$$

$$\frac{e \mapsto e'}{\mathbf{let} \bigcirc x = e\ \mathbf{in}\ e_0 \mapsto \mathbf{let} \bigcirc x = e'\ \mathbf{in}\ e_0}\ \text{(Op-}\bigcirc\text{-Tr)} \qquad \frac{}{\mathbf{let} \bigcirc x = \mathbf{return}\ v\ \mathbf{in}\ e' \mapsto e'\{v/x\}}\ \text{(Op-}\bigcirc\text{-Return-Tr)}$$

$$\frac{}{\mathbf{let} \bigcirc x = \{p, (e_i)_{i\in 1\ldots n}\}\ \mathbf{in}\ e' \mapsto \{p, (\mathbf{let} \bigcirc x = e_i\ \mathbf{in}\ e')_{i\in 1\ldots n}\}}\ \text{(Op-}\bigcirc\text{)} \qquad \frac{}{\mathbf{do}\ \{p, (e_i)_{i\in 1\ldots n}\} \mapsto \{p, (\mathbf{do}\ e_i)_{i\in 1\ldots n}\}}\ \text{(Op-Do)}$$

$$\frac{e \mapsto e'}{\mathbf{do}\ e \mapsto \mathbf{do}\ e'}\ \text{(Op-Do-Tr)} \qquad \frac{\emptyset \neq I \subseteq 1\ldots n \qquad s_i \mapsto s_i' \qquad s_j = s_j' = f_j \qquad (\forall i \in I, j \notin I)}{\{p, (s_i)_{i\in 1\ldots n}\} \mapsto \{p, (s_i')_{i\in 1\ldots n}\}}\ \text{(Op-P-State-Tr)}$$

Figure 8. Selected Evaluation Rules

3. If $\phi, i : \kappa; \Phi; \Gamma \vdash e : \sigma$, then, for every $\phi \vdash I : \kappa$, we have $\phi; \Phi\{I/i\}; \Gamma\{I/i\} \vdash e\{I/i\} : \sigma\{I/i\}$.
4. If $\phi, i : \kappa; \Phi; \Gamma \vdash s : \sigma$, then, for every $\phi \vdash I : \kappa$, we have $\phi; \Phi\{I/i\}; \Gamma\{I/i\} \vdash s\{I/i\} : \sigma\{I/i\}$.

Proof: (1) follows directly from the definition of constraint satisfiability; (2) follows by induction on the derivation proving $\phi, i : \kappa; \Phi \models \sigma \sqsubseteq \tau$, using Point 1 when needed. (3) follows by induction on the derivation proving $\phi, i : \kappa; \Phi; \Gamma \vdash e : \sigma$, using Point 2 when needed. (4) follows by induction on the derivation proving $\phi, i : \kappa; \Phi; \Gamma \vdash s : \sigma$, using Point 3 when needed. □

4.2 Type Soundness and Type Preservation

Using the properties on size and sensitivities annotations detailed in the previous section, we are now ready to prove the usual properties we want a programming language to enjoy: substitution, type preservation, and progress.

4.2.1 Theorem [Substitution]: If $\phi; \Phi; \Gamma, x : !_R\tau \vdash e : \sigma$ and $\phi; \Phi; \Delta \vdash g : \tau$, then $\phi; \Phi; \Gamma + R \cdot \Delta \vdash e\{g/x\} : \sigma$.

Proof: By induction on the derivation proving $\phi; \Phi; \Gamma, x : !_R\tau \vdash e : \sigma$. □

The proof of the Substitution Lemma is straightforward, except for the management of the term variable environments. The proof of Type Preservation, however, is not as straightforward because it requires managing the constraints and the size and sensitivity annotations in various places.

4.2.2 Theorem [Type Preservation]:

1. If $\vdash e : \sigma$ and $e \mapsto e'$, then $\vdash e' : \sigma$.
2. If $\vdash s : \sigma$ and $s \mapsto s'$, then $\vdash s' : \sigma$.

Proof: Part (1) proceeds by induction on the derivation proving $\vdash e : \sigma$ and case analysis on the possible derivations for $e \mapsto e'$, using the Substitution Lemma (4.2.1). Lemmas (4.1.1) and (4.1.2) are needed when the step taken comes from a dependent pattern matching rule. Part (2) now follows by induction on the derivation proving $\vdash s : \sigma$ and case analysis on the possible derivations for $s \mapsto s'$, using Point 1 when needed. □

We can also prove Progress as usual:

4.2.3 Theorem [Progress]:

1. If $\vdash e : \sigma$, then either $e \mapsto e'$, or e is a value.
2. If $\vdash s : \sigma$, then either $s \mapsto s'$, or s is final.

Proof: Both parts proceed by induction on the given derivation, using part (1) as needed in part (2). □

5. Metric Preservation and ϵ-Differential Privacy

The design of the *DFuzz* type system is intimately related to the metric relation we present in this section. This connection is captured by the Metric Preservation Theorem (5.2.7), which states that the evaluations of two well typed expressions at a given distance result in two values at the same distance or less.

5.1 Metric Relations

To formalize the notion of sensitivity, we need a metric relation on programs and states that captures an appropriate notion of "information distance" for each type. For this purpose, we first introduce a metric relation $\sim_r$ on values and final states, and then extend it to a metric relation $\approx_r$ on expressions and states through substitutions of related values. Concretely, we begin with metric judgments on values and final states

$$\vdash v \sim_r v' : \sigma \qquad \vdash f \sim_r f' : \sigma$$

asserting, respectively, that values v, v' and final states f, f' are related at type σ and that they are no more than r apart, where $r \in \mathbb{R}^{\geq 0}$.

Using these metric judgments, we can also relate substitutions of values for variables in an environment Γ. First, we need some notation. Let Γ° be the environment obtained from Γ as follows:

$$\Gamma^\circ = \{x_i : \sigma_i \mid x_i : !_{R_i}\sigma_i \in \Gamma\}$$

Suppose that γ is a vector of positive reals indexed by variables in $\mathrm{dom}(\Gamma)$, i.e. $\gamma = (x_1 := r_1, \ldots, x_n := r_n)$. Then, we also define a metric judgment with shape:

$$\vdash \delta \sim_\gamma \delta' : \Gamma^\circ$$

This asserts that the substitutions δ and δ' of values for the variables in $\mathrm{dom}(\Gamma)$ are related at the types described by Γ°, and that they are no more than γ apart. That is, for every value $v_i = \delta(x_i)$ and $v_i' = \delta'(x_i)$ we have $\vdash v_i \sim_{\gamma(x_i)} v_i'$.

Finally, we have judgments for expressions and states:

$$\Gamma \vdash e \approx_r e' : \sigma \qquad \Gamma \vdash s \approx_r s' : \sigma$$

These assert that the expressions e, e' and states s, s' are related at the type σ, and that they are no more then r apart, in the type environment Γ.

Given an environment $\Gamma = (x_1 : !_{R_1}\sigma_1, \ldots, x_n : !_{R_n}\sigma_n)$ and a variable-indexed vector of positive reals $\gamma = (x_1 := r_1, \ldots, x_n := r_n)$, we define $\gamma[\![\Gamma]\!]$ as $\sum_{i=1}^n r_i \cdot [\![R_i]\!]$. By definition, $\gamma[\![\Gamma]\!]$ can assume values in $\mathbb{R}^{\geq 0} \cup \{\infty\}$. In what follows, we will be particularly interested in the cases where $\gamma[\![\Gamma]\!]$ is finite.

All these metric judgments are defined inductively by the rules in Figure 9 where $|b_1 \triangle b_2|$ is the size of the symmetric difference between the two databases b_1 and b_2. It is worth noting that the metric on expressions considers only expressions that are typable with no constraints and no free size variables. This ensures that the index r in the relations $\sim_r$ and $\approx_r$ is actually a value $r \in \mathbb{R}^{\geq 0}$.

The metric presented here, like the one used in [33], differs significantly from the one presented in [34] in its treatment of non-value expressions and states. In particular, we do not require the relation on expressions to be closed under reduction. This makes the proof of metric preservation easier, and, as we will see, it is sufficient to ensure differential privacy.

5.2 Metric Preservation

The Metric Preservation Theorem (5.2.7), which we present at the end of this section, can be seen as an extension of the Type Preservation Theorem (4.2.2). We can read it as asserting that the evaluation of expressions and states preserves not only their types but also the distances between related input values, up to a constant factor given by the metric relation.

The proof of the metric preservation theorem involves five major steps. The goal of these steps is to ensure that the different metric relations respect the properties of the type system. The first step is to show that the metric on expressions and states internalizes a sort of weakening:

5.2.1 Lemma:

1. If $\Delta \vdash e \approx_r e' : \sigma$ and $r \leq p$, then $\Delta \vdash e \approx_p e' : \sigma$.
2. If $\Delta \vdash s \approx_r s' : \sigma$ and $r \leq p$, then $\Delta \vdash s \approx_p s' : \sigma$.

Proof: The proof of (1) is by inversion on the rule proving $\Delta \vdash e \approx_r e' : \sigma$, using the fact that, if $\vdash v \sim_{r'} v' : \tau$ and $r' \leq p'$, then $\vdash v \sim_{p'} v' : \tau$. The base case of (2) follows from (1); the inductive case follows directly by induction hypothesis. □

The second step is to show that the metric relation is well behaved with respect to the subtyping relation. This is formalized by the following lemma.

5.2.2 Lemma [Subtyping on metrics]:

1. If $\vdash e \approx_r e' : \sigma$, and $\emptyset;\emptyset \vdash \sigma \sqsubseteq \tau$, then $\vdash e \approx_r e' : \tau$.
2. If $\vdash s \approx_r s' : \sigma$, and $\emptyset;\emptyset \vdash \sigma \sqsubseteq \tau$, then $\vdash s \approx_r s' : \tau$.

Proof: (1) by induction on the derivation proving $\vdash e \approx_r e' : \sigma$; (2) by induction on the derivation proving $\vdash s \approx_r s' : \sigma$. □

The third technical lemma is an intermediate step to show that the two metric relations $\sim_r$ and $\approx_r$ coincide on expressions and states that happen to be values and final states, respectively.

5.2.3 Lemma:

1. Suppose $\emptyset;\emptyset;\Gamma \vdash e : \tau$. If $\vdash \delta_1 \sim_\gamma \delta_2 : \Gamma^\circ$ and $\delta_1 e$ is a value, then $\delta_2 e$ must also be a value, and $\vdash \delta_1 e \sim_{\gamma[\![\Gamma]\!]} \delta_2 e : \tau$.
2. Suppose $\emptyset;\emptyset;\Gamma \vdash s : \tau$. If $\vdash \delta_1 \sim_\gamma \delta_2 : \Gamma^\circ$ and $\delta_1 s$ is final, then $\delta_2 s$ must also be final, and $\vdash \delta_1 s \sim_{\gamma[\![\Gamma]\!]} \delta_2 s : \tau$.

5.2.4 Corollary:

1. $\vdash v \approx_r v' : \tau$ iff $\vdash v \sim_r v' : \tau$.
2. $\vdash f \approx_r f' : \tau$ iff $\vdash f \sim_r f' : \tau$.

The last important property that the metric inherits from the *DFuzz* type system is a substitution property on the judgments involving the relation $\approx_r$.

5.2.5 Lemma [Substitution into $\approx$]:

The following rule is admissible:

$$\frac{\Delta_1 \vdash e_1 \approx_{r_1} e_1' : \tau_1 \qquad \Delta_2, x' : !_R\tau_1 \vdash e_2 \approx_{r_2} e_2' : \tau_2}{R \cdot \Delta_1 + \Delta_2 \vdash e_2\{e_1/x\} \approx_{r_1[\![R]\!]+r_2} e_2'\{e_1'/x\} : \tau_2}$$

Combining these four results, we can prove the main lemma:

5.2.6 Lemma [Metric Compatibility]:

Suppose $\vdash \delta \sim_\gamma \delta' : \Gamma^\circ$ such that $\gamma[\![\Gamma]\!] \in \mathbb{R}^{\geq 0}$. Then:

1. If $\emptyset;\emptyset;\Gamma \vdash e : \sigma$ and $\delta e \mapsto g$, then $\exists g'.\ \delta' e \mapsto g'$ and $\vdash g \approx_{\gamma[\![\Gamma]\!]} g' : \sigma$.
2. If $\emptyset;\emptyset;\Gamma \vdash s : \sigma$ and $\delta s \mapsto s_f$, then $\exists s_f'.\ \delta' s \mapsto s_f'$ and $\vdash s_f \approx_{\gamma[\![\Gamma]\!]} s_f' : \sigma$.

Proof: By induction on the typing derivation proving $\emptyset;\emptyset;\Gamma \vdash e : \sigma$ and $\emptyset;\emptyset;\Gamma \vdash f : \sigma$, respectively, with further case analysis on the evaluation step taken and using Corollary (5.2.4) and the Substitution Lemma (5.2.5). □

The Metric Compatibility lemma is the main ingredient we need to prove that well-typed programs map related input values to related output values:

5.2.7 Theorem [Metric Preservation]:

1. If $\vdash e \approx_r e' : \sigma$ and $e \mapsto e_f$, then $\exists e_f'.\ e' \mapsto e_f'$ and $\vdash e_f \approx_r e_f' : \sigma$.
2. If $\vdash s \approx_r s' : \sigma$ and $s \mapsto s_f$, then $\exists s_f'.\ s' \mapsto s_f'$ and $\vdash s_f \approx_r s_f' : \sigma$.

Proof: (1) By inversion on the rule proving the judgment $\vdash e \approx_r e' : \sigma$, using the Metric Compatibility Lemma (5.2.6); (2) by induction on the derivation proving the judgment $\vdash s \approx_r s' : \sigma$, again using Lemma (5.2.6). □

We can then make a corresponding statement about the complete evaluation $\hookrightarrow$ of two expressions or two states, where $\hookrightarrow$ is the reflexive, transitive closure of the step relation $\mapsto$:

5.2.8 Theorem [Big-Step Metric Preservation]:

1. If $\vdash e \approx_r e' : \sigma$ and $e \hookrightarrow v$, then there exists v' such that $e' \hookrightarrow v'$ and $\vdash v \sim_r v' : \sigma$.
2. If $\vdash s \approx_r s' : \sigma$ and $s \hookrightarrow f$, then there exists f' such that $e' \hookrightarrow f'$ and $\vdash f \sim_r f' : \sigma$.

$$\frac{\emptyset;\emptyset;\emptyset \vdash v : \sigma}{\vdash v \sim_0 v : \sigma} \qquad \frac{\emptyset;\emptyset;\emptyset \vdash f : \sigma}{\vdash f \sim_0 f : \sigma} \qquad \frac{\emptyset;\emptyset;\emptyset \vdash n : \mathbf{N}[S]}{\vdash n \sim_0 n : \mathbf{N}[S]} \qquad \frac{\emptyset;\emptyset;\emptyset \vdash r : \mathbb{R}[R]}{\vdash r \sim_0 r : \mathbb{R}[R]} \qquad \frac{\emptyset;\emptyset;\emptyset \vdash \mathtt{nil} : \mathbf{L}(\sigma)[0]}{\vdash \mathtt{nil} \sim_0 \mathtt{nil} : \mathbf{L}(\sigma)[0]}$$

$$\frac{\vdash v_1 \sim_{r_1} v_1' : \sigma \quad \vdash v_2 \sim_{r_2} v_2' : \mathbf{L}(\sigma)[S]}{\vdash \mathtt{cons}[S](v_1, v_2) \sim_{r_1+r_2} \mathtt{cons}[S](v_1', v_2') : \mathbf{L}(\sigma)[S+1]} \qquad \frac{\vdash v \sim_r v' : \sigma \qquad r \leq r'}{\vdash v \sim_{r'} v' : \sigma} \qquad \frac{|r_1 - r_2| = r}{\vdash r_1 \sim_r r_2 : \mathbb{R}} \qquad \frac{|b_1 \triangle b_2| = r}{\vdash b_1 \sim_r b_2 : \mathbf{db}}$$

$$\frac{x : !_R\sigma \vdash e \approx_r e' : \tau}{\vdash \lambda x.e \sim_r \lambda x.e' : !_R\sigma \multimap \tau} \qquad \frac{x : !_\infty\sigma \vdash e \approx_r e' : \sigma}{\vdash \mathbf{fix}\, x.e \sim_r \mathbf{fix}\, x.e' : \sigma} \qquad \frac{\vdash v \sim_r v' : \sigma\{I/i\}}{\vdash \mathbf{pack}(v, I)\ \mathbf{as}\ \exists i : \kappa.\sigma \sim_r \mathbf{pack}(v', I)\ \mathbf{as}\ \exists i : \kappa.\sigma : \exists i : \kappa.\sigma}$$

$$\frac{\forall \vdash R : \kappa \quad \vdash v\{R/i\} \sim_r v'\{R/i\} : \sigma\{R/i\}}{\vdash \Lambda i.v \sim_r \Lambda i.v' : \forall i : \kappa.\sigma} \qquad \frac{|\ln(r_i/r_i')| \leq s \qquad (\forall i)}{\vdash (r_1, \ldots, r_n) \sim_s (r_1', \ldots, r_n') : \mathcal{P}[n]}$$

$$\frac{\vdash p \sim_r p' : \mathcal{P}[n] \qquad \vdash f_i \sim_s f_i' : \tau \quad (\forall i)}{\vdash \{p, (f_1, \ldots, f_n)\} \sim_{r+s} \{p', (f_1', \ldots, f_n')\} : \tau} \qquad \frac{\vdash p \sim_r p' : \mathcal{P}[n] \qquad \vdash e_i \approx_s e_i' : \bigcirc\tau \quad (\forall i)}{\vdash \{p, (e_1, \ldots, e_n)\} \sim_{r+s} \{p', (e_1', \ldots, e_n')\} : \bigcirc\tau}$$

$$\frac{\vdash p \sim_r p' : \mathcal{P}[n] \qquad \Gamma \vdash s_i \approx_s s_i' : \tau \quad (\forall i)}{\Gamma \vdash \{p, (s_1, \ldots, s_n)\} \approx_{r+s} \{p', (s_1', \ldots, s_n')\} : \tau} \qquad \frac{\Gamma \vdash e \approx_r e' : \bigcirc\tau}{\Gamma \vdash \mathbf{do}\, e \approx_r \mathbf{do}\, e' : \tau} \qquad \frac{\vdash \delta_1 \sim_\gamma \delta_2 : \Gamma^\circ \quad \emptyset;\emptyset;\Gamma,\Delta \vdash e : \sigma}{\Delta \vdash \delta_1 e \approx_{\gamma[\![\Gamma]\!]} \delta_2 e : \sigma}$$

$$\frac{\vdash v_i \sim_{r_i} v_i' : \sigma_i \quad (\forall i)}{\{v_1/x_1, \cdots, v_n/x_n\} \sim_{(x_1:=r_1, \cdots, x_n:=r_n)} \{v_1'/x_1, \cdots, v_n'/x_n\} : (x_1 : \sigma_1, \ldots, x_n : \sigma_n)}$$

Figure 9. Metric Rules

5.3 Well-Typed Programs are ϵ-Differentially Private

The Big-Step Metric Preservation Theorem (5.2.8) ensures that programs map related inputs to related outputs. Combined with the properties of the probability layer, this allow us to show that well-typed programs are differentially private.

To formalize this, we need to define the probability $\mathtt{Pr}_f[v]$ that a final state f yields a value v. We recursively define:

$$\mathtt{Pr}_{\mathbf{do\,return}\ v'}[v] = \begin{cases} 1 & \text{if } v = v' \\ 0 & \text{otherwise} \end{cases}$$

$$\mathtt{Pr}_{\{(p_1,\ldots,p_n),(f_1,\ldots,f_n)\}}[v] = \sum_{i=1}^{n} p_i\, \mathtt{Pr}_{f_i}[v]$$

Note that, by the typing rule for probabilistic states ($\{s\}$), the tuples $(p_1, \ldots, p_n), (f_1, \ldots, f_n)$ must have the same length. The metric on probability distributions is carefully chosen so that the metric on final states corresponds to the relation on probability distributions needed in the definition of differential privacy.

5.3.1 Lemma: Let $\vdash f : \sigma$ and $\vdash f' : \sigma$ be two closed final states such that $f \sim_r f' : \sigma$ for some $r \in \mathbb{R}^{\geq 0}$. Then, for every value $\vdash v : \sigma$,

$$\mathtt{Pr}_f[v] \leq e^r \mathtt{Pr}_{f'}[v].$$

Thus, we can show that the type system can verify that a program is ϵ-differentially private.

5.3.2 Theorem [ϵ-Differential Privacy]: The execution of any closed program e such that

$$\vdash e : !_\epsilon\sigma \multimap \bigcirc\tau$$

is an ϵ-differentially private function from σ to τ. That is, for all closed values $v, v' : \sigma$ such that $\vdash v \sim_r v' : \sigma$, and all closed values $w : \tau$, if $\mathbf{do}(e\ v) \hookrightarrow f$ and $\mathbf{do}(e\ v') \hookrightarrow f'$, then

$$\mathtt{Pr}_f[w] \leq e^{r\epsilon} \mathtt{Pr}_{f'}[w].$$

Proof: By using the fact that $\vdash \{v/x\} \sim_{(x:=r)} \{v'/x\} : (x : \sigma)$, we have that $\vdash \mathbf{do}(e\ x)\{v/x\} \approx_{r\epsilon} \mathbf{do}(e\ x)\{v'/x\} : \tau$. So, by the Big-step Metric Preservation Theorem (5.2.8), we obtain $f \approx_{r\epsilon} f' : \tau$. We conclude by Corollary (5.2.4) and Lemma (5.3.1). □

The above theorem shows that in order to ensure that the execution of a program e corresponds to an ϵ-differentially private randomized function from values in σ to values in τ, it is sufficient to check that the program e has a type of this form:

$$\vdash e : !_\epsilon\sigma \multimap \bigcirc\tau$$

5.4 Primitive Operations for Privacy

As outlined in Section 3, one important property of *DFuzz* is that it can be extended by means of primitive operations. In particular, we are interested in adding two basic building blocks, allowing us to build more involved differentially private examples. The first operation we add is the *Laplace mechanism* (Proposition (2.4)), with the following signature:

$$add\_noise : \forall \epsilon : v.\mathbb{R}[\epsilon] \to\ !_\epsilon\mathbb{R} \multimap \bigcirc\mathbb{R}$$

Note that, unlike the version presented in *Fuzz*, this primitive allows the level of noise (and thus, the level of privacy) to be specified by the user.

Another primitive operation that fits well in our framework is the *exponential mechanism* [28]

$$\begin{aligned} exp\_noise :& \forall s : v, \epsilon : v.\mathbb{R}[s] \to \mathbf{Bag}(\sigma) \to (\sigma \to\ !_s\mathbf{db} \multimap \mathbb{R}) \\ & \to \mathbb{R}[\epsilon] \to\ !_\epsilon\mathbf{db} \multimap \bigcirc\sigma \end{aligned} \tag{1}$$

where $\mathbf{Bag}(\sigma)$ is a primitive type representing a multiset of objects of type σ. The exponential mechanism takes a set of possible outputs, a *quality score* that assigns to each element in the range a number (depending on the database), and the database itself. The quality score is at most s-sensitive in the database, and here we allow this sensitivity to be passed in as a parameter. The algorithm privately outputs an element of the range that approximately maximizes the quality score.

6. Case Studies

To illustrate how *DFuzz*'s dependent types expand the range of programs that can be certified as differentially private, we now present four examples of practical algorithms that can be implemented in *DFuzz*, but not in *Fuzz*. Each algorithm is taken from a different paper from the differential privacy literature (specifically, [4, 18, 19, 27]). The first three algorithms rely on the following new feature that is enabled by *DFuzz*'s dependent types:

Iterative privacy: The ability to express a dependency between the total privacy cost of a function and a parameter that is chosen at runtime, such as a number of iterations.

The last example illustrates how allowing slightly more complex sensitivity annotations can increase the expressivity, and it also shows another important use for dependent types:

Privacy-utility tradeoff: The ability of a function to control its own privacy cost, e.g., by scaling the precision of an iterated operation to make the total cost independent of the number of iterations.

We are experimenting with a prototype implementation of *DFuzz*, and we present the examples in the actual syntax used by the prototype, an extension of *Fuzz* that closely follows the concrete syntax from Figure 2 (for instance, we write `sample x=e; e'` to denote $\mathbf{let}\,\bigcirc x = e\,\mathbf{in}\,e'$); we only omit instantiations of size/sensitivity terms for brevity. The examples also use some additional constructs, such as `(a,b)` for tensor products of types `a` and `b`, with associated primitive operations. Details of these extensions are available in [14].

6.1 Iterative Privacy

***k-means* [4]** Our first example (Figure 10) is k-means clustering, an algorithm from data mining that groups a set of data points into k clusters. Given a set of points and an initial guess for the k cluster centers, the algorithm iteratively refines the clusters by first associating each point with the closest center, and then moving each center to the middle of its associated points; the differentially private version ensures privacy by adding some noise to the refined centers. The k-means algorithm can be implemented in *DFuzz* as follows: Here, `iterate` is a caller-supplied procedure performing

```
function kmeans
  (iters : Nat[i]) (eps : num[e])
  (db :[3 * i * e] (num,num) bag)
  (centers : list(num,num)[j])
  (iterate : num[e] -> (num,num) bag -o[3*e]
    list(num,num)[j] -> Circle list(num,num)[j])
  : Circle list(num,num)[j]
  {
    case iters of
      0     => return centers
    | n + 1 => sample next_centers =
                    kmeans n eps db centers iterate;
               iterate eps db next_centers
  }
```

Figure 10. k-means in *DFuzz*

an update of the centers (details omitted for brevity).

Note that the sensitivity of `kmeans` in the database `db` depends on two other parameters: the number of iterations `[i]` and the privacy cost per iteration `[e]` that the analyst is willing to tolerate. This is enabled by the dependent types in *DFuzz*; in contrast, *Fuzz* is only able to typecheck a simplified version of k-means in which both parameters are hard-coded.

***Iterative Database Construction* [19]** Our second example (Figure 11) is an algorithm that can efficiently answer *exponentially* many queries with good accuracy. This is done by first constructing a public approximation of the private database, which can then be used to answer queries without further privacy cost. IDC builds the approximation iteratively: given an initial guess, it uses a *private distinguisher (PA)* to find a query that would be inaccurate on the current approximation and then applies a *database update algorithm (DUA)* to refine the approximation. PA and DUA are parameters of the algorithm, and the sensitivity of an IDC instance depends on the sensitivities of its PA. This can be expressed in *DFuzz* as follows: Several choices for PA and DUA have been proposed in the

```
function IDC
  (iter : Nat[i]) (eps : num[e])
  (db :[2 * i * e] db_type) (qs : query bag)
  (PA : (query bag) -> approx_db
      -> db_type -o[e] Circle query)
  (DUA : approx_db -> query -> num -> approx_db)
  (eval_q : query -> db_type -o[1] num)
  : Circle approx_db {
  case iter of
    0     => return init_approx
  | n + 1 =>
      sample approx = (IDC n eps db qs PA DUA);
      sample q = PA qs approx db;
      sample actual = add_noise eps (eval_q q db);
      return (DUA approx q actual)
  }
```

Figure 11. Iterative Database Function in *DFuzz*

literature; some can be written directly in *DFuzz* (e.g., the exponential distinguisher from [22]), and others can be added as trusted primitives (e.g., the sparse distinguisher from [35]). By contrast, a parametric IDC cannot be written in plain *Fuzz* because there is no way to express the dependency between the sensitivity of the PA/DUA and the sensitivity of the overall algorithm.

***Cumulative Distribution Function* [27]** Our third example (Figure 12) is an algorithm that computes the CDF function. Given a database of numeric records and a list of buckets defined by cutoff values, it computes the number of records in each bucket. [27] presents three variants of this algorithm with different privacy/utility tradeoffs, only one of which was previously supported in *Fuzz*. To understand why, consider the following version, implemented in *DFuzz*: Note that the sensitivity in this version depends

```
function cdf
  (eps : num[e]) (buckets : list(num)[i])
  (db :[i * e] num bag) : Circle list(num)[i]
  {
    case buckets of
      []       => return []
    | (x :: y) =>
        let (lt,gt) = bag_split (fun n => n<x) db;
        sample count = add_noise eps (bag_size lt);
        sample bigger = cdf eps y gt;
        return (count :: bigger)
  }
```

Figure 12. Cumulative Distribution Function in *DFuzz*

on the number of cutoff values or "buckets." Since *Fuzz* cannot capture such a dependency, this CDF variant is not just impossible to write in *Fuzz*—it cannot even be added as a trusted primitive, since there is no way to express its type signature. In contrast, *DFuzz* can directly support this program, and it could also support the last version in [27] with a small extension that we discuss in Section 8.

6.2 Privacy-Utility Tradeoff

Our fourth and most complex example shows how *DFuzz* enables programmers to write functions that control their own privacy cost, and it also illustrates how small extensions to the language for sensitivity annotations can further increase expressivity. The extension we use here introduces a new operation $R_1 \mathbin{\tilde{\oslash}} R_2$ on sensitivity terms R_1 and R_2 that provides a limited form of division. The interpretation is extended accordingly as follows:

$$\begin{aligned}
&[\![R_1 \mathbin{\tilde{\oslash}} R_2]\!]_\rho = \frac{r}{s} \quad \text{if } [\![R_1]\!]_\rho = r \wedge [\![R_2]\!]_\rho = s \wedge r, s \notin \{0, \infty\} \\
&[\![R_1 \mathbin{\tilde{\oslash}} R_2]\!]_\rho = 0 \quad \text{if } [\![R_1]\!]_\rho = 0 \vee [\![R_2]\!]_\rho = \infty \\
&[\![R_1 \mathbin{\tilde{\oslash}} R_2]\!]_\rho = \infty \quad \text{otherwise}
\end{aligned}$$

The behavior of the $\tilde{\oslash}$ operator is different from that of the ordinary division operator: $\tilde{\oslash}$ does not enjoy the usual properties of division with respect to multiplication and addition, i.e., it is *not* the inverse of multiplication, and it does *not* enjoy the usual distributivity laws. However, it does have two key properties. First, Lemma (3.1) still holds for the system that includes $\tilde{\oslash}$. If the usual division $\div$ were added instead, the interpretation would no longer be total because $r \div 0$ and $\infty \div \infty$ are in general undefined when $\div$ is the inverse of $\hat{\cdot}$ and enjoys the usual distributivity laws. The choice of $\tilde{\oslash}$ ensures the preservation of the metatheoretic results of Sections 4 and 5. Second, the interpretation of $\tilde{\oslash}$ satisfies the following inequality (the need for which will become apparent shortly) for every ρ:

$$[\![(R_1 \mathbin{\tilde{\oslash}} (R_2 + 1)) \cdot R_2]\!]_\rho \leq [\![R_1]\!]_\rho \tag{2}$$

In terms of $\tilde{\oslash}$, we can add a *safe division* primitive to the language for terms:

$$div : \forall i : \upsilon . \forall j : \upsilon . \mathbb{R}[i] \to \mathbb{R}[j+1] \to \mathbb{R}[i \mathbin{\tilde{\oslash}} (j+1)]$$

Note that the div operation is simply a restriction of the usual division to positive real numbers that we make total by restricting the domain of the denominator to reals greater than or equal to 1.

***k-medians* [18]** With this extension, we can implement our fourth example (Figure 13): an algorithm for *k-medians*, a classic problem in combinatorial optimization. Given a database V of locations, with distances between locations, a desired number k of (say) factories to build, and a private demand set $D \subseteq V$, the goal is to select a set $F \subseteq V$ of k locations to minimize the cost, defined to be the sum of the distances from each demand point to the closest factory. In the program, the distances are given implicitly via the *cost function*, which maps sets of factory locations to costs. The algorithm starts with an initial random choice F_0 of k locations and runs several iterations, each time using the exponential mechanism (Equation (1) in Section 5.4) to find the best location to swap for a location in the candidate set of factories. After building up a collection of candidate factory sets, the mechanism uses the exponential mechanism once more to choose the best configuration.

The helper function `kmedians_aux` runs the main loop, which repeatedly tries to improve the cost of a set of locations by replacing a location from the set with one that is not in the set. The outer `kmedians` function simply chooses an initial (random) set of factory locations and sets up the privacy and iteration constants.

The program uses several primitive functions for manipulating bags: `bagcontains` checks the membership in bags, `bagproduct` builds the Cartesian product of two bags, `bagadd` adds an element to a bag, `bagswap` swaps two element in a bag, and `bagselect` chooses subset of a given size from a bag uniformly at random. The helper function `score` measures how much a swap can improve the cost of a set of locations. It can also be written in *DFuzz* (see [14]).

The key challenge in implementing k-medians is that it scales the privacy level of the exponential mechanism to achieve a *constant* overall privacy cost, independent of the number of iterations. To derive the correct type for `kmedians`, *DFuzz* must prove that, in the call to the auxiliary function `kmedians_aux`, the sensitivity of `D` is at most `eps`. This involves checking a subtyping application that requires the following inequality between sensitivity terms:

$$((e \mathbin{\tilde{\oslash}} (i+1)) \mathbin{\tilde{\oslash}} (2 \cdot s + 1)) \cdot (2 \cdot s) \cdot i \;\leq\; e$$

This inequality follows from two applications of Equation (2). Thus, with the additional $\tilde{\oslash}$ operator, algorithms that scale their privacy cost depending on the number of iterations can be expressed and verified in *DFuzz*, using dependent types.

```
function kmedians_aux
  (iter : Nat[i]) (F0 : (loc bag)) (delta : num[s])
  (cost : (loc bag) -> (loc bag) -o[s] num)
  (eps : num[e]) (V : loc bag)
  (D :[e * (2 * s) * i] loc bag)
  : Circle (loc bag, (loc bag) bag)
  {
    case iter of
    | 0 => return (F0, (bag F0))
    | x + 1 =>
       sample pair = kmedians_aux x F0 delta cost
                                        eps V D;
       let (Fi, Fs) = pair;
       let (_, F_comp) =
          bagsplit (fun l => bagcontains V l) Fi;
       swaps = bagproduct Fi F_comp;
       sample best = exp_noise delta swaps
                       (score cost Fi)
                       (eps * (2 * delta)) D;
       let (a, b) = best;
       Fnew = bagswap a b Fi;
       return (Fnew, bagadd Fs Fnew)
  }

function kmedians
  (T : Nat[i]) (delta : num[s]) (V : loc bag)
  (cost : (loc bag) -> (loc bag) -o[s] num)
  (D :[2 * e] loc bag) (eps : num[e])
  (k : Nat[n]) : Circle (loc bag)
  {
    sample F1 = bagselect k V;
    eps' = div(div(eps,(T + 1)),(2 * delta)+1);
    sample result = kmedians_aux T F1
                      delta cost eps' V D;
    let (_, Fs) = result;
    exp_noise delta Fs cost eps D
  }
```

Figure 13. k-medians in *DFuzz*

7. Related Work

Differential privacy Our system provides differential privacy [12], one of the strongest privacy guarantees that has been proposed to date. Alternatives include randomization, l-diversity, and k-anonymity, which are generally less restrictive but can be vulnerable to certain attacks on privacy [24]. Differential privacy offers a provable bound on the amount of information an attacker can learn about any individual, even with access to auxiliary information.

PINQ [26] is an SQL-like differentially private query language embedded in C#; Airavat [36] is a MapReduce-based solution using a modified Java VM. Both PINQ and Airavat check privacy at runtime, while *DFuzz* uses a static check. The other previous work

in this area is *Fuzz* [20, 33, 34], on which *DFuzz* is based. *DFuzz* has a richer type system: while *Fuzz* uses linear types to track sensitivity, *DFuzz* uses a combination of linear indexed types and lightweight dependent types, which substantially expands the set of differentially private queries that can be certified successfully.

Another recent language-based solution is CertiPriv [3]. This is a machine-assisted framework—built on top of the Coq proof assistant—for reasoning about differential privacy from first principles. CertiPriv can certify even more queries than *DFuzz*, including queries that do not rely on standard building blocks such as the Laplace mechanism—indeed, it can be used to prove the correctness of the Laplace mechanism itself. However, this comes at the cost of much higher complexity and less automation, making CertiPriv more suitable for experts who are expanding the boundaries of differential privacy. In contrast, *DFuzz*'s certification is automatic, allowing it to target a broader class of potential users.

Other work in this area include Xu [39], who considered differential privacy in a distributed setting, using a probabilisitc process calculus, and Mohan et al. [29], who introduce a platform for private data analysis that optimizes error for certain queries.

Linear dependent types Linear types, inspired by Girard's linear logic [15], have emerged as key tools to support fine grained reasoning about resource management [37]. In this context, the idea of using indexed modalities $!_p$ for counting multiple uses of the same resource, as we do, emerged early on. Bounded Linear Logic [16] introduced modalities indexed by polynomial expressions. Those are similar to our sensitivity annotations with the essential difference that they are polynomials over natural numbers, while we consider polynomials over the reals augmented with ∞. This approach has been extended in [9] by constrained universal and existential quantifiers, providing mechanisms for polymorphism and abstraction over polynomial expressions similar to the ones made available by our quantifiers over size and sensitivity variables.

Different forms of lightweight dependent types form the basis for programming languages such as Dependent ML, ATS, and Epigram. Moreover, they have also started to be incorporated in more standard programming languages, such as Haskell [40]. In all these approaches the types can express richer properties of the program that can then be ensured automatically by type checking.

ATS [7] is a language that combines linear and dependent types. Its type system is enriched with a notion of resources that is a type-level representation of memory locations. The management of location resources uses a linear discipline. This aspect makes ATS useful for reasoning about properties of memory and pointers. However, linear types as used in ATS are not enough to reason about the sensitivity of programs.

Linear indexed types and lightweight dependent types have also been combined in [8] with the aim of providing a general framework for implicit complexity. The approach in [8] is similar in spirit to the one developed in the current paper; however, it considers only type-level terms that represent natural numbers, and its typing judgments are parametric in an equational program providing for the semantics of the type-level language. Moreover, [8] allows a further dependency between different modal operators that is not needed for our analysis. More recently, Krishnaswami et al. [23] proposed a language that combines linear (non-indexed) types with dependent types in order to provide program modules with a refined control of their state.

Related notions of privacy and sensitivity The study of database privacy and statistical databases has a long history. Recent work includes Dalvi, Ré, and Suciu's study of probabilistic database management systems [10], and Machanavajjhala et al.'s comparison of different notions of privacy with respect to real-world census data [25].

Quantitative Information Flow is concerned with how much one piece of a program can affect another, but measures this in terms of how many bits of entropy leak during one execution. While Differential Privacy and Quantitative Information Flow are clearly related concepts, the question of establishing formal relations between them has been approached only recently [1, 2].

Provenance analysis in databases tracks the input data actually used to compute a query's output, and is also capable of detecting that the same piece of data was used multiple times to produce a given answer [17].

Palamidessi and Stronati [31] recently proposed a constraint-based approach to compute the sensitivity of relational algebra queries. Their analysis is able in particular to compute the exact sensitivity for a wide range of queries. In contrast, the goal of our approach is to provide an upper bound on the sensitivity not only of relational queries but also for higher order functional programs.

Chaudhuri et al. in [6] study automatic program analyses that establish sensitivity (which they call robustness) of numerical programs. Their approach can also used to ensure differential privacy, although they do not study this application in depth. Our approach differs in that we ensure differential privacy through a logically motivated type system, rather than a program analysis.

8. Conclusions and Future Work

We have presented a new core language, *DFuzz*, for differentially private queries. Like its predecessor, *Fuzz*, *DFuzz* has a type system with strong metatheoretic properties, which guarantee that all queries of a certain type are differentially private. The key novelty in *DFuzz* is a lightweight form of linear dependent types that track size and sensitivity information; this considerably expands the range of programs that can be certified as differentially private. We have demonstrated the expressivity of *DFuzz* using four example algorithms from the differential privacy literature that can be implemented in *Fuzz* only in greatly simplified form, or not at all.

Prototype We are currently working on an algorithmic version of *DFuzz* and a prototype implementation. The key implementation challenge is related to checking the subtype relation. A natural approach to this problem is to generate numeric and logical constraints that would have to be satisfied in order to type the program, and to then pass these constraints to an external solver. In the case of *DFuzz*, some of the constraints are over real numbers, but they are of a form that is supported by the Z3 solver [11]. Since *DFuzz*'s additional constructs and annotations are erasable at runtime, a prototype can share a backend with *Fuzz*, and is thus able to take advantage of *Fuzz*'s defenses against side channels [20].

Possible extensions The languages for size and sensitivity annotations we have used in this paper are fairly restrictive, but this is merely a design choice, and not an inherent limitation of the approach. As we have shown in the k-medians example, allowing more complex size and sensitivity annotations can increase the expressivity, though it may also make the generated constraints harder to solve.

There are other simple increments to the annotation languages that would increase *DFuzz*'s expressivity. For example, [27] provides a higher-utility algorithm for the cumulative distribution function that uses a divide-and-conquer approach. The resulting sensitivity is proportional to the *logarithm* of the number of buckets. If logarithms of sizes were added to the annotation language, it would become possible to implement this algorithm. (Of course, this addition would again come at the cost of increasing the difficulty of constraint solving.) Another possible direction would be to include more general data types, as in [38], in addition to the precise types $\mathbf{L}(\sigma)[S]$ and $\mathbf{N}[S]$ we have focused on. For instance, the private Facility Location algorithm, as presented in [18], is similar

to k-medians, but, instead of a fixed number of factories to build, we are given a fixed cost per factory, and the goal is to minimize the total cost. This algorithm requires recursion over generalized trees, which we speculate could be implemented in *DFuzz* if support for sized versions of these data types were added.

Limitations and future work We designed *DFuzz* to certify algorithms for which the differential privacy property can essentially be proven with a rich compositional sensitivity analysis to show that the noise is drawn from an appropriate distribution. However, in the differential privacy literature there are algorithms whose analysis requires more involved reasoning. One example is the private version of the Unweighted Vertex Cover algorithm that was presented in [18]. One way to handle the analysis for this algorithm would be to use the probabilistic *relational* reasoning that is the basis of the CertiPriv framework [3]. The ability to reason about relations between different programs and data, i.e., about closely neighboring databases, seems crucial to this approach. However, by using the CertiPriv framework, one loses the ability to reason about differential privacy in an automated way. One natural way to preserve the automatic approach and to nevertheless expand the scope of the analysis would be to combine the approach of *DFuzz* with a limited form of relational reasoning.

Acknowledgments We thank Emilio Jesús Gallego Arias for many valuable comments on the final version of this work. We thank Mário S. Alvim, Loris D'Antoni, Sanjeev Khanna, Aaron Roth, and the anonymous reviewers for their helpful comments. This research was supported by ONR grants N00014-09-1-0770 and N00014-12-1-0757, and NSF grant #1065060. Marco Gaboardi was supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 272487.

References

[1] M. S. Alvim, M. E. Andres, K. Chatzikokolakis, and C. Palamidessi. On the relation between Differential Privacy and Quantitative Information Flow. In *Proc. ICALP*, 2011.

[2] G. Barthe and B. Köpf. Information-theoretic Bounds for Differentially Private Mechanisms. In *Proc. CSF*, 2011.

[3] G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proc. POPL*, 2012.

[4] A. Blum, C. Dwork, F. McSherry, and K. Nissim. Practical privacy: the SuLQ framework. In *Proc. PODS*, 2005.

[5] K. Chaudhuri, C. Monteleoni, and A. D. Sarwate. Differentially private empirical risk minimization. *J. Mach. Learn. Res.*, 12:1069–1109, July 2011.

[6] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. NavidPour. Proving programs robust. In *Proc. FSE*, 2011.

[7] C. Chen and H. Xi. Combining programming with theorem proving. In *Proc. ICFP*, pages 66–77, 2005.

[8] U. Dal Lago and M. Gaboardi. Linear dependent types and relative completeness. In IEEE LICS '11, pages 133–142, 2011.

[9] U. Dal Lago and M. Hofmann. Bounded linear logic, revisited. In *TLCA*, volume 5608 of *LNCS*, pages 80–94. Springer, 2009.

[10] N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.

[11] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, 2008.

[12] C. Dwork. Differential privacy. In *Proc. ICALP*, 2006.

[13] C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *Proc. TCC*, 2006.

[14] M. Gaboardi, A. Haeberlen, J. Hsu, A. Narayan, and B. C. Pierce. Linear dependent types for differential privacy (extended version). `http://privacy.cis.upenn.edu/papers/dfuzz-long.pdf`.

[15] J. Girard. Linear logic. *Theor. Comp. Sci.*, 50(1):1–102, 1987.

[16] J. Girard, A. Scedrov, and P. Scott. Bounded linear logic. *Theoretical Computer Science*, 97(1):1–66, 1992.

[17] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proc. PODS*, 2007.

[18] A. Gupta, K. Ligett, F. McSherry, A. Roth, and K. Talwar. Differentially private combinatorial optimization. In *Proc. SODA*, 2010.

[19] A. Gupta, A. Roth, and J. Ullman. Iterative constructions and private data release. In *Proc. TCC*, 2012.

[20] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proc. USENIX Security*, 2011.

[21] S. Hayashi. Singleton, union and intersection types for program extraction. In *Proc. TACS*, volume 526 of *LNCS*, pages 701–730. Springer Berlin / Heidelberg, 1991.

[22] S. P. Kasiviswanathan, H. K. Lee, K. Nissim, S. Raskhodnikova, and A. Smith. What can we learn privately? In *Proc. FOCS*, Oct. 2008.

[23] N. R. Krishnaswami, A. Turon, D. Dreyer, and D. Garg. Superficially substructural types. In *Proc. ICFP*, 2012.

[24] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *Proc. ICDE*, 2007.

[25] A. Machanavajjhala, D. Kifer, J. Abowd, J. Gehrke, and L. Vilhuber. Privacy: Theory meets practice on the map. In *Proc. ICDE*, 2008.

[26] F. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proc. SIGMOD*, 2009.

[27] F. McSherry and R. Mahajan. Differentially-private network trace analysis. In *Proc. SIGCOMM*, 2010.

[28] F. McSherry and K. Talwar. Mechanism design via differential privacy. In *Proc. FOCS*, 2007.

[29] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler. GUPT: privacy preserving data analysis made easy. In *Proc. SIGMOD*, 2012.

[30] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *Proc. IEEE S&P*, 2008.

[31] C. Palamidessi and M. Stronati. Differential privacy for relational algebra: Improving the sensitivity bounds via constraint systems. In *Proc. QAPLS*, volume 85 of *EPTCS*, pages 92–105, 2012.

[32] N. Ramsey and A. Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *Proc. POPL*, 2002.

[33] J. Reed, M. Gaboardi, and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy (extended version). `http://privacy.cis.upenn.edu/papers/fuzz-long.pdf`.

[34] J. Reed and B. C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *Proc. ICFP*, Sept. 2010.

[35] A. Roth and T. Roughgarden. The median mechanism: Interactive and efficient privacy with multiple queries. In *Proc. STOC*, 2010.

[36] I. Roy, S. T. V. Setty, A. Kilzer, V. Shmatikov, and E. Witchel. Airavat: security and privacy for MapReduce. In *Proc. NSDI*, 2010.

[37] D. Walker. *Advanced Topics in Types and Programming Languages*, chapter Substructural Type Systems. The MIT Press, 2005.

[38] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. POPL*, 1999.

[39] L. Xu. Modular reasoning about differential privacy in a probabilistic process calculus. Manuscript, 2012.

[40] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. M. es. Giving Haskell a promotion. In *Proc. TLDI*, 2012.