Programming Language Techniques for Differential Privacy

Gilles Barthe
IMDEA Software

Marco Gaboardi
Univ. of Dundee

Justin Hsu
Univ. of Pennsylvania

Benjamin Pierce
Univ. of Pennsylvania

Differential privacy is rigorous framework for stating and enforcing privacy guarantees on computations over sensitive data. Informally, differential privacy ensures that the presence or absence of a single individual in a database has only a negligible statistical effect on the computation's result. Many specific algorithms have been proved differentially private, but manually checking that a given program is differentially private can be subtle, tedious, or both. This approach becomes unfeasible when larger programs are considered.

This situation has motivated research in developing techniques for assisting programmers in building differentially private applications. We survey a range of approaches based on ideas from programming language research, discussing the formal guarantees that each of these approaches provides and showing how each can be used to ensure differential privacy in practice.

1. INTRODUCTION

An enormous amount of data is gathered in databases every day: hospital records, network flows, location data, power sensor readings, etc. This information has many potential good uses—e.g., for scientific or medical research, network monitoring, . . . —but much of it cannot be safely released due to privacy concerns. Protecting privacy is hard: experience has repeatedly shown that when owners of sensitive datasets release derived data, they often reveal more than intended. Even careful efforts to protect privacy often prove inadequate. A notable example is the Netflix prize competition, which released movie ratings from subscribers. Although the data was carefully anonymized, Narayanan and Shmatikov were later able to de-anonymize many of the private records [Narayanan and Shmatikov 2008].

Privacy breaches often occur when the owner of the dataset uses an incorrect threat model—e.g., they make wrong assumptions about the knowledge available to attackers. In the case of Netflix, Narayanan and Shmatikov had access to auxiliary data in the form of a public, unanonymized data set (from IMDB) that contained similar ratings. Such errors are difficult to prevent without reasoning about arbitrary information that could be (or later become) available to an attacker.

One way out of this dilemma is to make sure that every computation on sensitive data satisfies *differential privacy* [Dwork et al. 2006], a very strong guarantee: if an individual's data is used in a differentially private computation, the probability of any given result changes by at most a factor of e^{ϵ} (compared to the situation where this individual's data is not used), where ϵ is a parameter controlling the tradeoff between privacy and accuracy. Differential privacy impresses by the long list of assumptions it does *not* require: it is not necessary to know what information an attacker has, whether attackers are colluding, or what the attackers are looking for in particular. For this reason, it is becoming a gold standard for data privacy.

A typical way to ensure differential privacy is by adding some statistical noise to a program's result. The amount of noise that needs to be added in order to ensure the bound described by the privacy parameter ϵ depends on the particular program at hand. This has motivated a large body of work in algorithm design aimed at designing programs that are differentially private.

The design space of differential privacy is constrained also by the program's *utility*, i.e., the *accuracy* of the data analysis that the program performs. Having precise results information is indeed often the very reason why we are interested in running a program over some data. Fortunately, the algorithms community has developed algorithms that can ensure differential privacy and at the same time can achieve a good level of accuracy.

An important aspect of the theory of differential privacy is that it provides several *composition schemes* useful to compose different mechanisms. So, mechanisms become basic building blocks that can be assembled by composition schemes. This way of building differentially private programs is very attractive since it makes it easier to reason about differential privacy also for programmers that are not privacy experts. However, manually checking that these composed programs are differentially private can be both tedious and rather subtle.

For this reason, several tools have been proposed with the goal of assisting a programmer in checking whether a given program is differentially private or not. In this brief survey article, we summarize a range of approaches based on the use of logical and verification techniques to ensure differential privacy. In particular, we will present three: the first is based on the use of type systems for analyzing the sensitivity of a program and was initiated by Reed and Pierce [2010]; the second is based on the idea of relational verification in program logic and was initiated by Barthe et al. [2012]; the third mixes the use of type systems and relational verification by proposing a relational type system and has been developed by Barthe et al. [2015]. Along the way, we discuss some other developments that have built on these works to enlarge the space of programs that can be verified differential privacy and to make these approaches more effective.

Besides the formal methods approaches, there have been several other approaches aiming at building systems for support differential privacy. We will discuss some of them in the related work section.

We will begin this overview by presenting in the next section differential privacy in a formal way.

2. DIFFERENTIAL PRIVACY

Differential privacy [Dwork et al. 2006] is a notion of privacy-preserving data analysis that guarantees strong bounds on the increase in harm that a user I incurs as a result of participating in the analysis, even under worst-case assumptions. More precisely, differential privacy is a property of a program P asserting that for any two databases[1] $D, D' \in \mathrm{dB}$ differing only on the data of some individual I, and for any subset $S \subseteq \mathcal{R}$ of outputs, the probability that P releases an element of S on D is "almost the same" as the probability that P releases an element of S on D'. Quantitatively, "almost the same" is defined in term of two parameters $\epsilon, \delta \geq 0$ as follows.

Definition 2.1 (Differential privacy [Dwork et al. 2006]). A randomized program $P : \mathrm{dB} \to \mathcal{R}$ is (ϵ, δ)-differentially private for $\epsilon, \delta \geq 0$ if for every $D, D' \in \mathrm{dB}$ differing in one row and for any subset $S \subseteq \mathcal{R}$ we have

$$\Pr[P(D) \in S] \leq e^{\epsilon} \cdot \Pr[P(D') \in S] + \delta. \tag{1}$$

[1]For our presentation we will assume that a program's input is always a "database," a multiset of rows of the same type, where each individual's data is in a single row. More general definitions can also be considered.

The parameter ϵ gives a bound on the increase of the probability of any specific event if the individual I's data is included in the analysis (including any bad event that the individual may be worried about). The parameter δ can be thought of as the probability of failure in providing this bound; it is typically very small (often zero). For small ϵ, the factor e^{ϵ} can be thought of as $1+\epsilon$. Both these parameters are crucial: ϵ is the *privacy cost*, the larger ϵ is, the more information about individuals is potentially revealed by the result; δ allows trading off privacy and utility. However, as we will see in the following sections, not all the formal approaches to differential privacy have considered $\delta > 0$; some have focused only on ϵ. In these cases we sometimes write ϵ-differential privacy instead of $(\epsilon, 0)$-differential privacy.

A key concept in building differentially private algorithms is the notion of *sensitivity*:

Definition 2.2. A program $P : A \to B$ is *c-sensitive* for $c \in \mathbb{R}_{>0}$ if, for all $x, y \in A$ we have

$$d_B(P(x), P(y)) \leq c \cdot d_A(x, y).$$

where d_A and d_B are metrics on A and B respectively.

In other words, a c-sensitive program magnifies changes in its inputs by at most a factor of c. The definition of sensitivity requires some metric on the type of inputs and outputs. In the following, for two databases $D_1, D_2 \in \mathrm{dB}$ we will use a metric counting the number of individuals that differs in D_1 and D_2, and we will say that D_1 and D_2 are *adjacent* if they differ in only one individual.

Real-valued programs with limited sensitivity can be converted into ϵ-differentially private queries by using the *Laplace mechanism* [Dwork et al. 2006]. Here, we write $\mathrm{Lap}(a, \beta)$ to denote the Laplace distribution centered in a with scale parameter β, presented in Figure 1, whose probability density function is $\frac{1}{2\beta} e^{\frac{-|a-x|}{\beta}}$.

THEOREM 2.3 (LAPLACE MECHANISM). *Let $P : \mathrm{dB} \to \mathbb{R}$ be a c-sensitive deterministic program, and let $Q : \mathrm{dB} \to \mathbb{R}$ be the randomized program $Q = \lambda b.\ P(b) + N$, where N is a random variable distributed according to $\mathrm{Lap}(0, \frac{c}{\epsilon})$. Then Q is ϵ-differentially private.*

In other words, the Laplace mechanism converts the deterministic program P into a randomized program 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 P (larger c), the more noise must be added to P's result to preserve privacy.

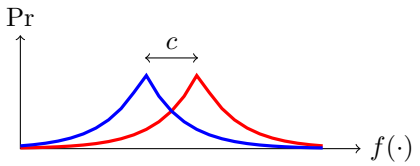

Fig. 1: Probability distributions of the Laplace mechanism for a c-sensitive function on two neighboring databases.

The Laplace mechanism ensures differential privacy while also providing a strong accuracy guarantee:

FACT 2.1. *Let $P : \mathrm{dB} \to \mathbb{R}$ be a c-sensitive deterministic program, and let $Q : \mathrm{dB} \to \mathbb{R}$ be the randomized program $Q = \lambda b.\ P(b) + N$, where N is a random variable distributed according to $\mathrm{Lap}(0, \frac{c}{\epsilon})$. Then, for every $\alpha \in (0, 1]$,*

$$\Pr\left[|Q(x) - P(x)| \geq \ln(\alpha^{-1}) \cdot \frac{c}{\epsilon}\right] \leq \alpha.$$

This can be visualized using the picture in Figure 1: with high probability, the Laplace distribution will return a value close to the original one.

The Laplace mechanism ensures ϵ-differential privacy for real-valued functions. Another way of ensuring ϵ-differential privacy for functions over arbitrary domains $\mathcal{R}$ is the *exponential mechanism* [McSherry and Talwar 2007]. Let $\mathcal{R}$ be a set of outputs to which we associate a *score function* $F : \mathrm{dB} \times \mathcal{R} \to \mathbb{R}$. Then, the exponential mechanism can be used to output an element of $\mathcal{R}$ that approximately maximizes the score function.

THEOREM 2.4 (EXPONENTIAL MECHANISM [MCSHERRY AND TALWAR 2007]). *Let* $F : \mathrm{dB} \times \mathcal{R} \to \mathbb{R}$ *be a* score function *that is* c*-sensitive in* dB*. The* exponential mechanism$\mathrm{Exp}(F, d, \epsilon)$ *takes as input* $d \in \mathrm{dB}$ *and returns* $r \in \mathcal{R}$ *with probability*

$$\frac{\exp(\epsilon F(d, r)/2c)}{\sum_{r' \in \mathcal{R}} \exp(\epsilon F(d, r')/2c)}.$$

This mechanism is ϵ*-differentially private.*

Several other mechanisms have been proposed for building differentially private programs; see the recent book by Dwork and Roth [2014] for some of them. These mechanisms can be composed using one of several composition principles offered by the theory of differential privacy. The most basic is this one:

THEOREM 2.5 (SEQUENTIAL COMPOSITION). *Let* P_1 *and* P_2 *be two differentially private programs with parameters* (ϵ_1, δ_1) *and* (ϵ_2, δ_2)*, respectively. Then* $\hat{P}(d) = (P_1(d), P_2(d))$ *is an* $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$*-differentially private program.*

This composition theorem is a strong property of differential privacy, and it also holds in an interactive setting where the program P_2 has access to the result of $P_1(d)$ (see Dwork et al. [2010]).

Another form of composition, formulated in terms of partitions of the database [McSherry 2009], is also useful in the design of practical programs.

THEOREM 2.6 (PARALLEL COMPOSITION). *Let* P *be a* $(\epsilon, delta)$*-differentially private program and let* (d_1, d_2) *be a partition of the database* d *in the two disjoint databases* d_1 *and* d_2*. Then the program*

$$\hat{P}(d) = (P(d_1), P(d_2))$$

is also (ϵ, δ)*-differentially private.*

Other more complex composition principles can also be used, e.g. the "advanced composition" principle from Dwork et al. [2010]. We will briefly discuss it in the conclusions.

Differential privacy has many other attractive formal properties. Interested readers can find more information in Dwork and Roth [2014].

3. TYPE SYSTEMS: THE FUZZ APPROACH

The first formal approach we introduce is based on the idea of using types to track the sensitivities of parts of a program and basic typed primitives to guarantee that the right amount of noise–proportional to the sensitivity—is added at appropriate points. Specifically, we present an approach proposed by Reed and Pierce [2010] that considers sensitivity as a resource and that uses *linear-indexed types* to explicitly track it. This approach has been implemented in the language ***Fuzz*** [Haeberlen et al. 2011].

The idea behind the design of ***Fuzz*** is to use the type-checking procedure to ensure that well-typed programs of a particular type are ϵ-differentially private. (***Fuzz*** does

$$\begin{array}{lll} \tau, \sigma & ::= & \mathbb{R} \mid !_r\sigma \multimap \tau \mid \sigma \oplus \tau \mid \mathsf{M}\ \sigma & \text{(types, } r \in \mathbb{R}^{\infty}_{>0}\text{)} \\ e & ::= & x \mid \mathrm{r} \mid \mathrm{f} \mid \lambda x :!_r\sigma.\ e \mid e\ e \mid \mathbf{fix}\, g.\, x.\, e \mid \mathbf{inj}_i\ e \mid \mathbf{case}\, e\, \mathbf{of}\, x \to e\, \mathbf{or}\, y \to e \\ & & \mathbf{let}\ \mathsf{M}\ x = e\ \mathbf{in}\ e \mid \mathbf{unit}\ e & \text{(expressions)} \\ \Gamma & ::= & \emptyset \mid \Gamma, x :!_r\sigma & \text{(environments)} \end{array}$$

Fig. 2: *Fuzz* syntax

not support (ϵ, δ)-differential privacy yet, though work in this direction is underway.) In particular, to ensure ϵ-differential privacy, an expression e must be typed as follows:

$$\vdash e \ : \ !_\epsilon \mathrm{dB} \multimap \mathsf{M}\ \mathcal{R} \tag{2}$$

This type has three key components: first, the type $\mathsf{M}\ \mathcal{R}$ representing the monadic type of discrete probability distributions over the output type R; second, the linear implication $\multimap$ representing the space of 1-sensitive functions from $!_\epsilon \mathrm{dB}$ to $\mathsf{M}\ R$; third, the type $!_\epsilon \mathrm{dB}$ representing the type of databases whose metric is that of dB multiplied by ϵ, where the *modality* $!_\epsilon$ is indexed by the scaling factor ϵ. Thus, the inhabitants of the type $!_\epsilon \mathrm{dB} \multimap \mathsf{M}\ \mathcal{R}$ are $epsilon$-sensitive programs mapping databases to probability distributions.

As a concrete instance, the term

$$\lambda d :!_\epsilon \mathrm{dB}.\ \mathit{add\_noise}(\mathit{count}(d)) \tag{3}$$

is a differentially private program that, given a database d, returns the number of elements in it, using the counting function *count* plus some noise from the Laplace distribution. We can give this program the type $!_\epsilon \mathrm{dB} \multimap \mathsf{M}\ \mathbb{R}$.

The fact that the type signature in (2) actually ensures differential privacy follows from the soundness of the *Fuzz* type system. To explain this in more detail, we need some additional formal preliminaries.

Figure 2 shows the formal grammar of a simplified core of *Fuzz*—a functional language with real-number constants, functions, fixpoints, conditionals (**case**), and the constructors of the monad M (**let** and **unit**). Real numbers appear in *Fuzz* in two places: as constants and as type annotations. The types are a linear refinement of the classical simply typed lambda-calculus, with some extra annotations for tracking the sensitivity of functions and with the monad M encapsulating probabilistic computations. In a function with type $!_r\sigma \multimap \tau$, the annotation r (drawn from $\mathbb{R}^{\infty}_{>0}$, the set of positive reals extended with ∞) gives an upper bound on the function's sensitivity. When r is ∞, it means that the sensitivity is not bounded. We define $r + \infty = \infty$, and $r \cdot \infty = \infty$. We write $\sigma \to \tau$ as a shorthand for $!_\infty\sigma \multimap \tau$.

The expressiveness of *Fuzz* can be enhanced by giving special typing rules to some standard arithmetic functions like addition and multiplication by a scalar; we elide this here. *Fuzz* can also be extended by other linear type operators like $\otimes$ and $\&$, and by other algebraic data types [Reed and Pierce 2010]. Interestingly, to each of these components one can associate a different metric.

Figure 3 shows the typing rules for this subset of *Fuzz*. The judgment $\Gamma \vdash e : \sigma$ can be read as "the expression e has type σ under the assumptions in environment Γ," where Γ records both the type of each free variable x appearing in e and an upper bound on the sensitivity of e to changes in x. Formally, this is achieved by using environments containing variable assignments of the form $x :!_r\tau$. The intuitive meaning is that e can be assigned type σ assuming x has type τ, and moreover that e is r-sensitive in x.

Sensitivities are managed via operations on typing environments. Given two environments Γ and Δ, we define the sum operation $\Gamma + \Delta$ as

$$\begin{array}{rcll} \Gamma + \Delta & = & \{x :!_{r_1+r_2}\sigma & \mid x :!_{r_1}\sigma \in \Gamma \wedge x :!_{r_2}\sigma \in \Delta\} \\ & \cup & \{x :!_{r_1}\sigma & \mid x :!_{r_1}\sigma \in \Gamma \wedge x \notin \mathrm{dom}(\Delta)\} \\ & \cup & \{x :!_{r_2}\sigma & \mid x \notin \mathrm{dom}(\Gamma) \wedge x :!_{r_2}\sigma \in \Delta\}. \end{array}$$

Given $r \in \mathbb{R}^{\infty}_{>0}$ and an environment Γ, the product operation $r \cdot \Gamma$ is defined as

$$r \cdot \Gamma = \{x :!_{r \cdot r_1}\sigma \mid x :!_{r_1}\sigma \in \Gamma\}.$$

The (Const) rule assigns the type $\mathbb{R}$ to a real number constant r; since r does not depend on the variables in Γ, the rule does not require further assumptions. The (Prim) rule assigns primitive functions their corresponding predefined types. The (Var) rule says that the variable x has type τ, if x is assigned the type τ by the environment and if the sensitivity annotation on x is at least 1 (since the value of x is 1-sensitive to changes in x). The $(\multimap I)$ rule says that if e is r-sensitive in the free variable x, and if e yields a value of type σ when x is of type τ, then the lambda-abstraction $\lambda x :!_r \tau.e$ is an r-sensitive function of type $!_r\tau \multimap \sigma$.

The most interesting rule is $(\multimap E)$: if e_1 is an r-sensitive function from τ to σ, and if e_2 has type τ, then 1) the application $e_1\ e_2$ has type σ, and 2) the sensitivity of $e_1\ e_2$ in each free variable x is r times the sensitivity of e_2 in x plus the sensitivity of e_1 in x (because each use of x in e_2 is "magnified" r times by the use that e_1 makes of its argument). Similar reasoning applies in the (Case) rule; another point specific to this rule is that the expressions e_l and e_r are required to have the *same* sensitivity annotations in their free variables. One way to enforce this constraint is taking for each variable the maximum of the sensitivity they have in e_l and e_r. Rule (inj_i) is for typing the disjoint sum injections. Rule (unit) is for typing the monadic unit; notice that the sensitivities in the environment are multiplied by ∞, because **unit** applied to two different values represents two distributions that (as we will see) are at distance ∞. The rule (let) is used to assign a type to the composition of two monadic computations. The composition corresponds to sampling from the first computation and using the obtained value in the second one. Notice that the environment in the conclusion of this rule is the sum of the two environments from the premises; in particular, the ∞ on the variable associated with the sample does not appear in the result; this because the distance of the result cannot depend on the sampled value. The fact that the sensitivities in the environments are summed corresponds to the sequential composition principle from Theorem 2.5. (The parallel composition theorem can also be internalized by using a tensor product, omitted here. See Reed and Pierce [2010].)

The remaining constructors for expressions can be introduced as typed combinators. For example the fixpoint combinator fix can be typed as $((!_r\tau \multimap \sigma) \to (!_r\tau \multimap \sigma)) \to (!_r\tau \multimap \sigma)$. (The fixpoint itself has unbounded sensitivity, but its return type is an r-sensitive function $!_r\tau \multimap \sigma$.) Other primitives can be added as described by the rule (Prim).

The operational semantics for *Fuzz* can be described via a standard small step evaluation semantics (which we elide). The evaluation steps are described by judgments of the shape $e \mapsto e'$, asserting that the expression e evaluates to the expression e'.

To connect the type system with the operational semantics and justify the way the rules propagate sensitivities, each type τ is equipped with a metric defining a "distance" between values that is also lifted to expressions. A *metric judgment* of the form $\vdash e_1 \approx_m e_2 : \tau$ indicates that the expressions e_1 and e_2 are related at type τ, and that they are no more than distance m apart with respect to the metric on τ (where $m \in \mathbb{R}^{\infty}_{\geq 0}$ and $\infty + m = \infty$, $0 \cdot \infty = 0$, and $m \cdot \infty = \infty$ for $m \neq 0$). The metric on the base type $\mathbb{R}$ is the standard distance metric on reals. Metrics for type constructors

$$\frac{}{\Gamma \vdash \mathrm{r} : \mathbb{R}}\ (\text{Const}) \qquad \frac{\mathrm{f}\ \text{an } r\text{-sensitive fn in } \sigma \to \tau}{\Gamma \vdash \mathrm{f}\ :\ !_r\sigma \multimap \tau}\ (\text{Prim}) \qquad \frac{r \geq 1}{\Gamma, x :!_r\tau \vdash x : \tau}\ (\text{Var})$$

$$\frac{\Gamma, x :!_r\tau \vdash e : \sigma}{\Gamma \vdash \lambda x :!_r\tau.\, e\ :\ !_r\tau \multimap \sigma}\ (\multimap I) \qquad \frac{\Gamma \vdash e_1 :!_r\tau \multimap \sigma \qquad \Delta \vdash e_2 : \tau}{\Gamma + r \cdot \Delta \vdash e_1\ e_2 : \sigma}\ (\multimap E)$$

$$\frac{\Gamma \vdash e : \tau_i}{\Gamma \vdash \mathbf{inj}_i\ e : \tau_1 \oplus \tau_2}\ (\mathrm{inj_i}) \qquad \frac{\Delta \vdash e : \sigma \oplus \tau \qquad x :!_r\sigma, \Gamma \vdash e_l : \mu \qquad y :!_r\tau, \Gamma \vdash e_r : \mu}{\Gamma + r \cdot \Delta \vdash \mathbf{case}\, e\, \mathbf{of}\, x \to e_l\ \mathbf{or}\ y \to e_r : \mu}\ (\text{Case})$$

$$\frac{\Gamma \vdash e : \sigma}{\infty \cdot \Gamma \vdash \mathbf{unit}\ e : \mathsf{M}\ \sigma}\ (\text{unit}) \qquad \frac{\Delta \vdash e : \mathsf{M}\ \sigma \qquad x :!_\infty\sigma, \Gamma \vdash e_1 : \mathsf{M}\ \mu}{\Gamma + \Delta \vdash \mathbf{let}\ \mathsf{M}\ x = e\ \mathbf{in}\ e_1 : \mathsf{M}\ \mu}\ (\text{let})$$

Fig. 3: Type assignment rules for *Fuzz*

like function types are defined in terms of the metrics on their components by logical relations (see Reed and Pierce [2010]). As an example we just provide here the metric on a type $\mathsf{M}\ \sigma$, representing a notion of distance between probabilities. This is defined as

$$\vdash e_1 \approx_m e_2 : \mathsf{M}\ \sigma \ \text{ iff } \ \max_{v:\sigma} \ln\left(\frac{\Pr[e_1 = v]}{\Pr[e_2 = v]}\right) \leq m.$$

Notice that this corresponds to the differential privacy requirement.

The *soundness* of the type system is stated with respect to the metric: every expression of type $!_r\tau \to \sigma$ actually computes a r-sensitive function from τ to σ.

THEOREM 3.1 (METRIC PRESERVATION). *Let* $\vdash e :!_r\tau \to \sigma$ *and* $\vdash v_1 \approx_m v_2 : \tau$. *If* $e\ v_1 \mapsto e_1$ *and* $e\ v_2 \mapsto e_2$, *then* $\vdash e_1 \approx_{r \cdot m} e_2$.

The proof of this theorem uses a metric step-indexed logical relation [Reed and Pierce 2010]. Using this theorem we can now state the formal connection between *Fuzz* typing and differential privacy.

COROLLARY 3.2 (DIFFERENTIAL PRIVACY). *If*

$$\vdash e :!_\epsilon \mathrm{dB} \multimap \mathsf{M}\ A$$

in Fuzz *then* e *is* ϵ*-differentially private.*

An example in *Fuzz*

Consider the simple query *"How many patients at this hospital are over the age of 40?"*, and suppose that we have the following functions available:

$$\begin{aligned} &\textit{over\_40} : \mathrm{row} \to \mathbb{B} \\ &\textit{size} : \mathrm{dB} \multimap \mathbb{R} \\ &\textit{filter} : (\mathrm{row} \to \mathbb{B}) \to \mathrm{dB} \multimap \mathrm{dB} \end{aligned}$$

The predicate *over_40* (which can be programmed in *Fuzz*) simply determines whether or not an individual database row indicates that the corresponding patient is over the age of 40. The primitive function *size* takes an entire database, and outputs how many rows it contains; its type records the fact that this operation is 1-sensitive. The higher-order primitive function *filter* takes a predicate on database rows and a database; it returns the subset of the rows in the database that satisfy the predicate. This filtering operation also has a sensitivity of 1 in its database argument.

With these functions in place, the query can be written as the program

$$\lambda d :!_\epsilon \mathrm{dB}.\ \textit{add\_noise}\ (\textit{size}\ (\textit{filter over\_40}\ d)) \quad : \quad !_\epsilon \mathrm{dB} \multimap \mathsf{M}\,\mathbb{R},$$

where *add_noise* is the primitive adding noise according to the Laplace distribution.

Dependent types

As presented so far, *Fuzz* provides a useful analysis only for programs where sensitivities are statically bounded. When the sensitivity depends on program inputs, *Fuzz* can only show that the program is ∞-sensitive, a trivial bound. This is a serious limitation: many private mechanisms use iteration, and by composition, the total privacy level (i.e., function sensitivity) is often a function of the number of iterations. Typically this number controls the accuracy-privacy tradeoff of the computation, and it is an input to the program.

To address this problem Gaboardi et al. [2013] proposed *DFuzz*, which can be seen as an extended version of the *Fuzz* type system allowing a lightweight form of dependent typing akin to *sized types*. The idea is to have types reflect size information of certain forms; $\mathsf{nat}[2]$ is the type of the natural number 2; $\mathsf{list}(\tau)[3]$ is the type of lists of length 3 with elements of type τ; $\mathsf{real}[0.5]$ is the type of the real number 0.5. (Notions of sized types can be defined for general inductive types, but to demonstrate the idea it is enough to consider natural numbers and lists, along with a sized type for reals.) Size information can also involve variables, known as *size variables*. These variables can describe relationships between the sizes of different expressions, even when the size is not statically specified. For instance, a function for reversing a list of booleans could have the type

$$\forall i.\ \mathsf{list}(\mathsf{bool})[i] \to \mathsf{list}(\mathsf{bool})[i],$$

indicating that the function preserves the length of the list, represented by the size variable i. Besides allowing simple size variables and constants as annotations, *DFuzz* features a small language of *size expressions* S:

$$S ::= i \mid n \mid S + S$$

With size annotations, we can reflect the sizes of the input parameters in the types. For instance, an input parameter representing the number of iterations can be given type $\mathsf{nat}[i]$. To connect this information with the sensitivity, *DFuzz* defines a language of *sensitivity expressions* R:

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

These annotations generalize the sensitivities in *Fuzz*: they can be real constants (as in *Fuzz*), or they can be variables, size expressions, or arithmetic combinations of other sensitivity expressions. By enriching the sensitivity language, *DFuzz* is able to model sensitivities that are not known statically. For instance, a function that has sensitivity 3 times the input parameter—possibly representing the number of iterations—is reflected by the *DFuzz* type $\forall i.\ !_{3 \cdot i}\mathsf{nat}[i] \multimap \mathsf{real}$.

The inclusion of size variables requires additional bookkeeping in the judgments. First, judgments track the size and sensitivity variables that are in scope with a *index variable context* ϕ. Second, judgments can also record equality statements about size expressions. This information is crucial for recording information in case analyses on sized types. For instance, when we perform a case on an expression with type $\mathsf{nat}[S]$, the two cases should record our assumptions about the size expression S: in the first case, $S = 0$, while in the second case, $S = i + 1$ for a fresh size variable i. This information is stored in the *constraint context* Φ, which is a conjunction of equalities $S_1 = S_2$.

```
function cdf (eps : num[e]) (buckets : list(num)[i])
             (b :[i * e] dB) : M list(num)[i]{
    case buckets of
      []       => unit []
    | (x :: y) =>
        res = filter (fun n => n<x) b;
        let M count = add_noise eps (size res);
        recstep = cdf eps y b;
        unit (count :: recstep)
  }
```

Fig. 4: Pseudo-code for the Cumulative Distribution Function in *DFuzz*

Putting everything together, *DFuzz* judgments have the following form:

$$\phi; \Phi; \Gamma \vdash e \ : \ \tau.$$

The rules of the type system for *DFuzz* are an extension of the ones for *Fuzz*. As an example, here is the rule for conditionals:

$$\frac{\begin{array}{c}\phi; \Phi; \Gamma \vdash e : \mathsf{list}(\tau)[S] \qquad \phi; \Phi \wedge S = 0; \Delta \vdash u_1 : \sigma \\ \phi, i : \iota; \Phi \wedge S = i + 1; \Delta, x :!_R \mathsf{list}(\tau)[i], y :!_R \tau \vdash u_2 : \sigma\end{array}}{\phi; \Phi; \Delta + R \cdot \Gamma \vdash \mathbf{case}\, e\, \mathbf{of}\, \mathsf{nil} \to u_1\, \mathbf{or}\, \mathbf{cons}(y, x[i]) \to u_2 : \sigma} \ (\text{Dcase})$$

DFuzz's expressive power also comes from the availability of a subtyping relation (for sizes and sensitivities) and the availability of universal and existential quantifiers over size and sensitivity variables (see Gaboardi et al. [2013]). Finally, *DFuzz* enjoys a *metric preservation* theorem analogous to Theorem 3.1, ensuring differential privacy.

An example in *DFuzz*

We show how to use *DFuzz* to type an algorithm that computes the Cumulative Distribution Function (CDF) as presented by McSherry and Mahajan [2010]. Given a database of numeric records and a list of "buckets" defined by cutoff values, it computes the number of records in each bucket. There are several variants of this algorithm with different privacy/utility tradeoffs. We consider the one counting the number of record in each bucket and adding independent noise to each of them.

We show how this algorithm can be written in *Fuzz* concrete syntax in Figure 4. We use several basic data types and primitive extending the ones we used in Section 3. The list of buckets has type `list(num)[i]` asserting that it consists of a list of numbers of length `i`, for a size variable `i`. The return type is similar, except for the presence of the monad `M` asserting that the output is a distribution over lists of numbers of length `i`. The epsilon parameter `eps` has a precise type `num[e]` asserting that it is equal to the sensitivity variable `e`. We omitted the binder for size and sensitivity variables for keeping the notation lighter but both of them are quantified universally. The database is encoded with the type `dB` and we assume that a database is a multiset or bag of numbers. On this datatype we also have a filter operation `filter` that remove all the database elements that do not satisfy a given predicate. We use this predicate to collect only the elements that are in a particular bucket.

The bulk of the code is an iteration on the list of buckets of the filter operation with predicates that depends on each bucket value followed by counting and adding noise. Notice that because of this iteration the sensitivity depends on the number of buckets. For this reason, the database variable `b` in the type signature has the sensitivity annotation `[i*e]` asserting that overall `cdf` is `i*e`-differentially private. To ensure this

typing the type-checker uses assumptions about the sensitivity of the test when typing the `case` constructor, as described by the rule (Dcase) presented above. Notice that in *Fuzz* this example cannot be proved differentially private because of this dependency.

Further developments

The approach proposed by *Fuzz* has been extended in different directions. Eigner and Maffei [2013] extend the type system of *Fuzz* to consider a notion of local state and to integrate ideas from type systems for cryptographic protocols. This extension has been used to verify differential private concrete distributed protocols. D'Antoni et al. [2013] provide an automated sensitivity inference tool for *Fuzz* where the actual type-inference process is built on top of a familiar ML-like language by using SMT solvers to handle the numerical annotations. Using a similar approach, Azevedo de Amorim et al. [2014] study instead the type checking problem for *DFuzz*.

4. RELATIONAL HOARE LOGIC: THE CERTIPRIV APPROACH

The second approach that we summarize is based on the idea of using *relational Hoare logic* for verifying differential privacy, as implemented in the tool `CertiPriv` [Barthe et al. 2012]. In traditional Hoare logic one can reason about a program using logical predicates as pre- and post-conditions in judgments of the form

$$\vdash c : \phi \Longrightarrow \psi.$$

Intuitively (and ignoring termination issues), this judgment expresses the fact that given a memory m satisfying the logical predicate ϕ (the precondition), the command c will produce a memory m' satisfying the logical predicate ψ (the postcondition). In relational Hoare logic, one instead reasons about *two* programs, with logical relations as pre- and post-conditions. This approach was first introduced by Benton [2004] as a way to prove the correctness of compiler optimizations for a core (deterministic) imperative language. This approach can be made formal by using judgments of the form

$$\vdash c_1 \sim c_2 : \Psi \Longrightarrow \Phi.$$

Intuitively, this judgment expresses the fact that given two memories m_1, m_2 satisfying the logical relation Ψ (the precondition), the command c_1 and c_2 will produce memories m_1' and m_2' satisfying the logical relation Φ (the postcondition).

The logic proposed by Barthe et al. [2012] is an *approximate*, *probabilistic*, and *relational* Hoare logic, named apRHL, for reasoning about differential privacy. The *approximate* in apRHL refers to the fact that in this logic one can prove statements about approximate equivalence between probabilistic distributions—in other words one can reason in a principled way about notions of distance over distributions.

Judgments in apRHL are of the form[2]

$$\vdash c_1 \sim_{\langle \epsilon, \delta \rangle} c_2 : \Psi \Longrightarrow \Phi, \tag{4}$$

where c_1 and c_2 are probabilistic programs, Ψ and Φ are *relations* over memories, and ϵ, δ are real values representing the parameters of differential privacy and describing some notion of distance over the distributions computed by the probabilistic programs c_1 and c_2.

When c is a probabilistic program and $c_\triangleleft$ and $c_\triangleright$ are two copies of c resulting from renaming the variables in c in two different ways, one can use judgments of this form to assert differential privacy. The reading of this judgment is that, if

$$\vdash c_\triangleleft \sim_{\langle \epsilon, \delta \rangle} c_\triangleright : \Psi \Longrightarrow \mathtt{out}_\triangleleft = \mathtt{out}_\triangleright$$

[2]The original apRHL rules are based on a multiplicative privacy budget. We adapt the rules to an additive privacy parameter for consistency with the rest of the article.

$$\frac{\vdash c_1 \sim_{\langle\epsilon,\delta\rangle} c_2 : \Psi \wedge b_1 \Longrightarrow \Phi \quad \vdash d_1 \sim_{\langle\epsilon,\delta\rangle} d_2 : \Psi \wedge \neg b_1 \Longrightarrow \Phi}{\vdash \mathsf{if}\, b_1 \,\mathsf{then}\, c_1 \,\mathsf{else}\, d_1 \sim_{\langle\epsilon,\delta\rangle} \mathsf{if}\, b_2 \,\mathsf{then}\, c_2 \,\mathsf{else}\, d_2 : \Psi \wedge b_1 = b_2 \Longrightarrow \Phi}[\mathsf{cond}]$$

$$\frac{\vdash c_1 \sim_{\langle\epsilon,\delta\rangle} c_2 : \Theta \wedge b_1 \wedge k = e \Longrightarrow \Theta \wedge k < e \qquad \Theta \wedge n \leq e \implies \neg b_1 \qquad \Theta \implies b_1 = b_2}{\vdash \mathsf{while}\, b_1 \,\mathsf{do}\, c_1 \sim_{\langle n\epsilon, n\delta\rangle} \mathsf{while}\, b_2 \,\mathsf{do}\, c_2 : \Theta \wedge 0 \leq e \Longrightarrow \Theta \wedge \neg b_1}[\mathsf{while}]$$

$$\frac{\vdash c_1 \sim_{\langle\epsilon,\delta\rangle} c_2 : \Psi \Longrightarrow \Phi' \quad \vdash c_1' \sim_{\langle\epsilon',\delta'\rangle} c_2' : \Phi' \Longrightarrow \Phi}{\vdash c_1; c_1' \sim_{\langle\epsilon+\epsilon',\delta+\delta'\rangle} c_2; c_2' : \Psi \Longrightarrow \Phi}[\mathsf{seq}]$$

$$\frac{}{\vdash y_1 \xleftarrow{\$} \mathsf{Lap}_\epsilon(e_1) \sim_{\langle|e_1-e_2|\epsilon,0\rangle} y_2 \xleftarrow{\$} \mathsf{Lap}_\epsilon(e_2) : \mathsf{true} \Longrightarrow y_1 = y_2}[\mathsf{lap}]$$

$$\frac{}{\vdash y_1 \xleftarrow{\$} \mathsf{Exp}_\epsilon(s_1, e_1) \sim_{\langle\epsilon \max_r |s_1(x_1,r)-s_2(x_2,r)|,0\rangle} y_2 \xleftarrow{\$} \mathsf{Exp}_{\epsilon,s}(s_2, e_2) : s_1 = s_2 \Longrightarrow y_1 = y_2}[\mathsf{exp}]$$

Fig. 5: Selected proof rules of apRHL

---

is derivable, then c is (ϵ, δ)-differentially private with respect to the relation Ψ on initial memories—here $\mathtt{out}$ denotes the output value of c, so $\mathtt{out}_\triangleleft$ and $\mathtt{out}_\triangleright$ respectively denote the outputs of the first and second runs of c. In particular, the relation Ψ can be the adjacency condition over databases as in the usual definition of differential privacy.

The fact that the judgment of Equation 4 actually ensures differential privacy follow from the soundness of the logic. In order to express the notion of valid judgment and hence to state soundness of the logic, we first need some more formal preliminaries.

apRHL expresses judgments on pWHILE commands defined by the following grammar:

$$c ::= \mathsf{skip} \mid c;\, c \mid x \leftarrow e \mid x \xleftarrow{\$} \mathsf{Lap}_\epsilon(e) \mid x \xleftarrow{\$} \mathsf{Exp}_\epsilon(e, e) \mid \mathsf{if}\, e \,\mathsf{then}\, c \,\mathsf{else}\, c \mid \mathsf{while}\, e \,\mathsf{do}\, c$$

where x is a variable and e is an expression drawn from a language including simply typed lambda terms and basic operations on booleans, lists and integers. The only non-standard commands are the probabilistic assignments involving $\mathsf{Lap}_\epsilon(e)$ and $\mathsf{Exp}_\epsilon(e, e)$ which internalize the (discrete version of the) mechanisms of Theorem 2.3 and Theorem 2.4 with parameter ϵ.

The original presentation of the apRHL logic [Barthe et al. 2012] is organized in three sets of rules: the first set includes a set of core rules, the second set includes rules for mechanisms such as the Laplace and Exponential mechanisms, the third one includes generalized rules for loops. We omit the latter and we present a selection of the first two sets of rules in Fig. 5. Before introducing *validity* for judgments in apRHL we first need to define the semantics of pWHILE programs.

The semantics of pWHILE is probabilistic. To describe probabilities we consider the set $\mathsf{M}\ A$ of *sub-distributions* over a set A. This is the set of functions $\mu : A \to [0, 1]$ with discrete support such that $\sum_{x \in A} \mu\, x \leq 1$; when equality holds, μ is a true *distribution*. Sub-distributions can be given the structure of a monad[3]: for every function $g : A \to \mathsf{M}\ B$ and distribution $\mu : \mathsf{M}\ A$, we define $g^\star\, \mu : \mathsf{M}\ B$ to be the sub-distribution

$$g^\star\, \mu \stackrel{\text{def}}{=} \lambda b.\ \sum_{a \in A} (g\, a\, b)(\mu\, a),$$

The unit of the monad is given by the function $\lambda a.\mathbf{1}_a : A \to \mathsf{M}\ A$ that given an element $a \in A$ returns a probability distribution with all the mass assigned to the value a.

[3]That is why we use the notation $\mathsf{M}\ A$.

The semantics of a well-typed pWHILE program is defined by its (probabilistic) action on *memories*; we denote the set of memories by $\mathcal{M}$. A program memory $m \in \mathcal{M}$ is a partial assignment of values to variables. Formally, the semantics of a pWHILE program c is a function $[\![c]\!] : \mathcal{M} \to \mathsf{M}\,\mathcal{M}$ mapping a memory $m \in \mathcal{M}$ to a distribution $[\![c]\!]\,m \in \mathsf{M}\,\mathcal{M}$. The denotational semantics of programs is largely standard and we provide here as an example the interpretation of some selected constructions:

$$[\![\mathsf{skip}]\!]\,m = \mathbf{1}_m \qquad [\![c_1;\, c_2]\!]\,m = [\![c_2]\!]^\star\,([\![c_1]\!]\,m)$$
$$[\![\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]\!]\,m = \mathbf{if}\ ([\![e]\!]\,m = \mathsf{true})\ \mathbf{then}\ ([\![c_1]\!]\,m)\ \mathbf{else}\ ([\![c_2]\!]\,m)$$

skip is interpreted using the unit of the monad while the sequential composition ; is interpreted by using the composition of the monad. We also show the interpretation of the conditional to illustrate that the interpretation of commands requires also an interpretation of expressions $[\![e]\!] : \mathcal{M} \to \mathcal{M}$.

It will be convenient to use an alternative characterization of (ϵ, δ)-differential privacy based on the notion of ϵ-distance. Given $\mu_1, \mu_2 \in \mathsf{M}\ A$, the ϵ-distance Δ_ϵ is defined as

$$\Delta_\epsilon(\mu_1, \mu_2) \ \stackrel{\text{def}}{=}\ \max_{S \subseteq A} (\mu_1\, S - e^\epsilon\, \mu_2\, S),$$

where $\mu\, S \stackrel{\text{def}}{=} \sum_{a \in S} \mu\, a$. Note that $\Delta_\epsilon(\mu_1, \mu_2) \geq 0$. By the definition of ϵ-distance, a probabilistic program c is (ϵ, δ)-differentially private with respect to $\epsilon > 0$, $\delta \geq 0$, if for every two adjacent memories m_1 and m_2, we have

$$\Delta_\epsilon([\![c]\!]\ m_1, [\![c]\!]\ m_2) \leq \delta.$$

The validity in apRHL can be described in terms of a particular monadic lifting $\mathcal{L}_{\epsilon,\delta}(\cdot)$ that turns a relation $\Psi \subseteq \mathcal{M} \times \mathcal{M}$ over memories into a relation $\mathcal{L}_{\epsilon,\delta}(\Psi)\ \mathsf{M}\ \mathcal{M} \times \mathsf{M}\ \mathcal{M}$, where the lifting is defined as:

Definition 4.1 (*Lifting of a relation*). Given $\Psi \subseteq \mathcal{M} \times \mathcal{M}$, we have $\mathcal{L}_{\epsilon,\delta}(\Psi)\ \ \mu_1\ \mu_2$ iff there is a distribution $\mu \in \mathsf{M}\ (\mathcal{M} \times \mathcal{M})$ such that

(1) $\mu\,(a, b) > 0$ implies $(a, b) \in \Psi$,
(2) $\pi_1\,\mu \leq \mu_1$ and $\pi_2\,\mu \leq \mu_2$, and
(3) $\Delta_\epsilon(\mu_1, \pi_1\,\mu) \leq \delta$ and $\Delta_\epsilon(\mu_2, \pi_2\,\mu) \leq \delta$,

where $\pi_1\,\mu = \lambda x. \sum_y \mu\,(x, y)$ and $\pi_2\,\mu = \lambda y. \sum_x \mu\,(x, y)$.

This notion of lifting generalizes the notion of coupling, used in probability theory to represent a possible joint distribution μ of μ_1 and μ_2, see Barthe et al. [2015] for more general uses of this lifting relation. We can now define validity in apRHL as follows.

Definition 4.2 (*Validity in* apRHL). A judgment $\vdash c_1 \sim_{\langle \epsilon,\delta \rangle} c_2 : \Psi \Longrightarrow \Phi$ is valid iff

$$\forall m_1, m_2.\ m_1 \Psi m_2 \Longrightarrow \mathcal{L}_{\epsilon,\delta}(\Psi)\ \ ([\![c_1]\!]m_1)([\![c_2]\!]m_2)$$

Using this notion of validity, the definition of lifting and the definition of ϵ distance one can prove the following theorem stating the correctness for differential privacy.

THEOREM 4.3 (DIFFERENTIAL PRIVACY IN apRHL).
Let c be a pWHILE *command, if*

$$\vdash c_\triangleleft \sim_{\langle \epsilon,\delta \rangle} c_\triangleright : \mathrm{adjacent} \Longrightarrow =$$

Then, c is (ϵ, δ)-differentially private.

An example in CertiPriv

As an example, we describe a differentially private algorithm for computing partial sums over lists of integer values taken from a bounded interval $[0, M]$; the example is a mild generalization of the one presented by Chan et al. [2011], which considers the case $M = 1$ for privately computing more elaborate statistics over streams, including for instance heavy hitters. Given a list a of length n, the goal of the algorithm is to release private approximations of the partial sums $c[j] = \sum_{i=0}^{j} a[i]$, for all $j \leq n$. The two most immediate approaches for achieving this goal are:

(1) input perturbation: add Laplace noise to each element of the list and compute all partial sums using the noised list;
(2) output perturbation: compute partial sums accurately and add Laplace noise to each partial sum.

However, the first approach offers poor privacy and good accuracy, and the second one offers poor accuracy but good privacy. A more suitable solution, which achieves a reasonable trade-off between privacy and accuracy, combines these two approaches, using the more private but less accurate approach for computing partial sums on chunks of the list, and the less private but more accurate approach for computing partial sums of the list from the partial sums on chunks. More specifically, we assume that the list a has length $n = q \cdot m$. The algorithm is shown in Figure 6, using array notation.

```
SmartSum (a: list(int))(epsilon:R): list(real)
  j = 0; s = 0; x = 0;
  while (j < n){
      x = Lap(epsilon,a[j]);
      if j mod q = 0 {c[j] = c[j-q] + s + x; s = 0;}
      else {s = s  + a[j]; c[j]= c[j-1] + x;}
  return c;
```

Fig. 6: CertiPriv code for computing partial sums

The privacy analysis of the algorithm can be established in three steps. First, one performs code rewriting to obtain a semantically equivalent program which uses explicitly the two naive solutions: output perturbation and input perturbation. Then, one proves that each of these algorithms is differentially private. Finally, one uses the rule for loops to conclude that the smart sum algorithm achieves 2ϵ-differential privacy, when $M = 1$. For other values of M, the noise must be proportional to M in order to achieve privacy. In fact, the rule for loops we presented in Figure 5 yields an overly conservative privacy bound, so a more advanced rule is also needed to prove 2ϵ-privacy, see Barthe et al. [2012].

Further developments

Barthe et al. [2013] have used EasyCrypt—a tool for verifying cryptographic protocols based on a logic similar to CertiPriv—to verify that programs are *computational differential private*. This is a computational analogous of the standard definition of differential privacy [Mironov et al. 2009]. Barthe et al. [2014] have developed an approach to reduce the derivability in apRHL to derivability in standard Hoare Logic and use the latter to verify differential privacy. This can be done by duplicating part of the code of the original program and by using ghost variables to keep track of the privacy budget.

5. RELATIONAL TYPE SYSTEMS: THE HOARE^2 APPROACH

The third approach we present combines the advantages of the two approaches we presented before: the higher order approach of *Fuzz* with the expressiveness of apRHL. This approach is based on the use of *Higher Order Approximate Refinement* types and it has been implemented in the tool HOARe^2 [Barthe et al. 2015].

The idea of refinement types is to specify properties of expressions with types of the form $\{x :: \sigma \mid \Phi\}$. This should be read as: "x is a variable of type σ, satisfying the logical formula Φ". These assertions serve two purposes: (1) they express facts about the code (both the whole program and subprograms) and (2) they assert mathematical facts about primitive operations. A refinement type system formally verify that the first kind of annotations are correct, while assuming the assertions of the second kind as axioms.

Similarly to apRHL, in HOARe^2 the assertion Φ is *relational*: it can refer to two "copies" of each variable x, usually written $x_\triangleleft$ and $x_\triangleright$. The idea is that one may make assertions about two runs of the same program, where in the first run one can use the variable $x_\triangleleft$, and in the second one the variable $x_\triangleright$. For instance, the type $\{x :: \mathbb{N} \mid |x_\triangleleft - x_\triangleright| \leq k\}$ models pairs of natural numbers which differ by at most k. Relational refinements can be used to model (ϵ, δ)-differentially private computations as follows:

$$\vdash e : \{x :: \mathrm{dB} \mid \mathrm{adjacent}(x_\triangleleft, x_\triangleright)\} \to \mathsf{M}_{\epsilon,\delta}[\{y :: U \mid y_\triangleleft = y_\triangleright\}], \tag{5}$$

This typing judgment can be read as e is a program that run over two adjacent databases $x_\triangleleft$ and $x_\triangleright$ returns as outputs two probability distributions satisfying the definition of differential privacy. In particular, the output type uses once again a monad for probabilistic computations. Now, however the monad is parametrized by ϵ and δ, and the underlying type is a relational refinement stating that in the two runs one needs to consider only the probability of returning the same value—as required by the definition of differential privacy. Once again the fact that the typing judgment in Equation 5 actually ensures differential privacy follows from the soundness of the type system and we will now give more details to understand it.

HOARe^2 is a relational type discipline for a λ-calculus with probabilities, inductive types and recursion. For simplicity, here we will consider a terminating fragment[4] of HOARe^2, we refer to Barthe et al. [2015] for the presentation of the full system.

Most of the syntax of expression is standard and similar to the one of *Fuzz* presented in Section 3. However, HOARe^2 also have a set of *relational expressions* built over the set of *relational* variables. These are variables x with associated a left instance $x_\triangleleft$ and a right instance $x_\triangleright$. Expressions are used in the subject of typing judgments, and correspond to the actual programs to which one can assign semantics. Relational expressions are used in assertions that can appear in refinement types. Similarly, HOARe^2 distinguishes *simple types*, expressing properties of a single interpretation of an expression, and *relational types*, which express properties about two interpretations of an expression. Relational types extend the grammar of simple types with relational refinements, and use a dependent function type rather than standard function types. Simple types are standard while relational types are elements of the following grammar

$$T, U \in \mathcal{T} ::= \widetilde{\tau} \mid \mathsf{M}_{\epsilon,\delta}[T] \mid \Pi(x :: T).\, T \mid \{x :: T \mid \phi\}$$

where $\widetilde{\tau}$ is a basic type. The type $\Pi(x :: T).\, U$ corresponds to the dependent type product of T over U indexed by x. A type of the shape $\{x :: T \mid \Phi\}$ *refines* the type T using the assertion Φ. In both these types the bound variable is required to be relational, to bound both the left instance $x_\triangleleft$ and the right instance $x_\triangleright$ of the bound variable x. Assertions Φ are built from primitive assertions using the standard connectives and quantification

[4] In fact, HOARe^2 doesn't require termination but uses a partiality monad to track possibly unbounded recursions. This ensures the consistency of the system even in presence of higher-order refinement types.

$$\frac{(d_1,d_2)\in[\![\widetilde{\tau}]\!]^2}{(d_1,d_2)\in(\!|\widetilde{\tau}|\!)_\theta} \qquad \frac{(d_1,d_2)\in(\!|T|\!)_\theta \qquad [\![\phi]\!]_{\theta\left\{\begin{smallmatrix}x_\triangleleft\mapsto d_1\\ x_\triangleright\mapsto d_2\end{smallmatrix}\right\}}}{(d_1,d_2)\in(\!|\{x::T\mid\phi\}|\!)_\theta} \qquad \frac{\mu_1,\mu_2\in\mathsf{M}[|T|] \quad \mathcal{L}_{\epsilon,\delta}((\!|T|\!)_\theta)\ \mu_1\ \mu_2}{(\mu_1,\mu_2)\in(\!|\mathsf{M}_{\epsilon,\delta}[T]|\!)_\theta}$$

$$\frac{(f_1,f_2)\in[\![|T|\to|U|]\!]^2 \qquad \forall(d_1,d_2)\in(\!|T|\!)_\theta.\,(f_1(d_1),f_2(d_2))\in(\!|U|\!)_{\theta\{x_\triangleleft\mapsto d_1,x_\triangleright\mapsto d_2\}}}{(f_1,f_2)\in(\!|\Pi(x::T).\,U|\!)_\theta}$$

Fig. 7: Relational interpretation of types

over both relational and standard variables. Primitive assertions are equalities and inequalities over relational expressions.

The simply typed part of HOARe^2 is standard and omitted here. Simply typed terms are interpreted using a denotational semantics that is also largely standard. In particular, similarly to what happens in apRHL, a simply typed monadic type $\mathsf{M}\,\hat{T}$ is interpreted as the space of discrete probability distributions over elements in the interpretation of the simple type $\hat{T}$. The relationally typed part of HOARe^2 is less standard and requires several ingredients. A *relational environment* $\mathcal{G}$ is a finite map defined by relational bindings $(x::T)$ s.t. a variable is never bound twice and only relational variables are bound. We write $x\mathcal{G}$ for the application of the finite map $\mathcal{G}$ to x.

A *valuation* θ is a map from variables to the interpretation of valus. Given a valuation θ we can interpret relational assertions and relational types. Assertions are interpreted relationally in the expected way where some care is needed for quantifiers since the interpretation distinguishes between binders for relational and standard variables.

Relational types are interpreted as sets of pairs of elements of the interpretation of the erased type. The formal interpretation is given in 7 where $|\cdot|$ is a type erasure from relational to simple types, which maps dependent products to function spaces, and erases refinements and the indexes of the probabilistic monad. The definition of $|\cdot|$ extends recursively to relational environments.

Notice that a pair (d_1,d_2) is in the relational interpretation of a refinement $\{x::T\mid\phi\}$ if the assertion ϕ holds in a relational context where d_1 and d_2 are assigned to $x_\triangleleft$ and $x_\triangleright$, respectively. The relational interpretation of the dependent product is defined in a *logical relation* style: it relates function elements f_1, f_2 that map related elements d_1, d_2 (in $(\!|T|\!)_\theta$) to related elements (in $(\!|U|\!)_{\theta\{x_\triangleleft\mapsto d_1,x_\triangleright\mapsto d_2\}}$).

The monadic type $(\!|\mathsf{M}_{\epsilon,\delta}[T]|\!)_\theta$ is interpreted using a lifting construction $\mathcal{L}_{\epsilon,\delta}(\cdot)$ that turns a relation Ψ on $T_1\times T_2$ into a relation $\mathcal{L}_{\epsilon,\delta}(\Psi)$ on $\mathsf{M}\,T_1\times\mathsf{M}\,T_2$, where the lifting is defined similarly to the one presented in Definition 4.1.

The relational typing rules proves judgments of the shape $\mathcal{G}\vdash e_1\sim e_2::T$ expressing the fact that e_1 and e_2 are well typed at the relational type T under the relational environment $\mathcal{G}$. We give a selection of the typing rule in Figure 8 where we use $\mathcal{G}\vdash e::T$ as a shorthand for $\mathcal{G}\vdash e\sim e::T$.

As in relational Hoare logic [Benton 2004], one can distinguish between 1-sided and 2-sided rules; the latter operate on both expressions of the judgments, whereas the former operate on a single expression and can relate expressions that have different shapes.

The `case` construction is an example of a rule with both a 1-side and a 2-side version. The 1-side rule requires a synchronicity condition: the same branch must be taken in the left and right expressions. For the case of lists, this is ensured by requiring that the matched lists are either both empty or both non-empty. In contrast, the 2-sided rule

$$\text{C}\ \frac{\begin{array}{c}\mathcal{G} \vdash T \qquad \mathcal{G} \vdash e :: \widetilde{\tau}\ \texttt{list} \qquad \forall \theta.\, \theta \vdash \mathcal{G} \implies [\![(|e|_\triangleleft = \epsilon) \Leftrightarrow (|e|_\triangleright = \epsilon)]\!]_\theta \\ \mathcal{G} \vdash e_1 :: T \qquad \mathcal{G}, x :: \widetilde{\tau}, y :: \widetilde{\tau}\ \texttt{list}, \{|e|_\triangleleft = x_\triangleleft :: y_\triangleleft \wedge |e|_\triangleright = x_\triangleright :: y_\triangleright\} \vdash e_2 :: T\end{array}}{\mathcal{G} \vdash \texttt{case}\ e\ \texttt{with}\ [\epsilon \Rightarrow e_1 \mid x :: y \Rightarrow e_2] :: T}$$

$$\text{M}\ \frac{\begin{array}{c}\mathcal{G} \vdash e_1 :: \mathsf{M}_{\epsilon_1,\delta_1}[T_1] \\ \mathcal{G} \vdash \mathsf{M}_{\epsilon_2,\delta_2}[T_2] \qquad \mathcal{G}, x :: T_1 \vdash e_2 :: \mathsf{M}_{\epsilon_2,\delta_2}[T_2]\end{array}}{\mathcal{G} \vdash \textbf{let}\ \mathsf{M}\ x = e_1\ \texttt{in}\ e_2 :: \mathsf{M}_{\epsilon_1+\epsilon_2,\delta_1+\delta_2}[T_2]} \qquad \text{AR}\ \frac{e_1 \to e_1' \qquad \mathcal{G} \vdash e_1 \sim e_2 :: T}{\mathcal{G} \vdash e_1' \sim e_2 :: T}$$

$$\text{AC}\ \frac{\begin{array}{c}\mathcal{G} \vdash T \qquad |\mathcal{G}| \vdash e : \widetilde{\tau}\ \texttt{list} \qquad |\mathcal{G}| \vdash e' : |T| \\ \mathcal{G}, \{|e|_\triangleleft = \epsilon\} \vdash e_1 \sim e' :: T \qquad \mathcal{G}, x :: \widetilde{\tau}, y :: \widetilde{\tau}\ \texttt{list}, \{|e|_\triangleleft = x_\triangleleft :: y_\triangleleft\} \vdash e_2 \sim e' :: T\end{array}}{\mathcal{G} \vdash \texttt{case}\ e\ \texttt{with}\ [\epsilon \Rightarrow e_1 \mid x :: y \Rightarrow e_2] \sim e' :: T}$$

Fig. 8: Relational Typing (Selected Rules)

does not require this condition. The reduction rule (AR) close typing under reduction, and is useful to relate expressions that do not have the same shape.

The main property of the refinement typing is the soundness with respect to the denotational semantics. This is expressed by the following theorem.

THEOREM 5.1 (SOUNDNESS). *If $\mathcal{G} \vdash e_1 \sim e_2 :: T$, then for every valuation θ validating $\mathcal{G}$ we have $([\![e_1]\!]_\theta, [\![e_2]\!]_\theta) \in (\!|T|\!)_\theta$.*

It follows that HOARe^2 accurately models differential privacy.

COROLLARY 5.2 (DIFFERENTIAL PRIVACY). *If*

$$\vdash e :: \{x :: \mathrm{dB} \mid \mathrm{adjacent}(x_\triangleleft, x_\triangleright)\} \to \mathsf{M}_{\epsilon,\delta}[\{y :: \tau \mid y_\triangleleft = y_\triangleright\}]$$

then $[\![e]\!]$ is (ϵ, δ)-differentially private.

HOARe^2 is very expressive and in particular all the (terminating) programs of *DFuzz* can be embedded in it. To do this, the first step is to define by using assertions a predicate $\mathcal{D}_\sigma$ for each type. Then, one can define an map $(-)^*$ on types, terms and contexts of *DFuzz* turning them in corresponding components of HOARe^2 ensuring similar invariants. One then have the following:

THEOREM 5.3 (*DFuzz* EMBEDDING). *If $\phi; \Phi; \Gamma \vdash_D e : \tau$ then*

$$\phi^*, \Phi^*, \Gamma^* \vdash e^* :: \{y :: \tau^* \mid \mathcal{D}_\tau(y_\triangleleft, y_\triangleright) \leq \mathcal{D}_\Gamma(\Gamma_\triangleleft^*, \Gamma_\triangleright^*)\},$$

where $\mathcal{D}_\Gamma(\Gamma_\triangleleft^*, \Gamma_\triangleright^*) = \sum_{x :\, !_R \sigma \in \Gamma} R \cdot \mathcal{D}_\sigma(x_\triangleleft, x_\triangleright)$.

The main insight to prove this theorem is that one can define the metric of DFuzz—which is defined by logical relations—by using refinements. Then, the type system permits to preserve this invariant for the whole translation.

An example in HOARe^2

As an example we describe here an algorithm for privately answering a large set of queries. The Laplace mechanism is a simple solution, but it's known that this will add noise to each query proportional to $\sqrt{k}$ for k queries under (ϵ, δ)-privacy. When k is large, the large noise will make the released answers completely useless. Fortunately, there is a line of algorithms where noise is added in a carefully correlated manner, guaranteeing privacy while adding noise proportional only to $\log k$. We present one such algorithm, called DualQuery [Gaboardi et al. 2014]. The algorithm is parameterized

by a natural number s and a set qs of queries to answer accurately. The input is the number of rounds t and database b, and the output is a private synthetic database that is accurate for the given queries. The code of the algorithm is in Figure 9. We encode the

```
Function DualQuery ({t::num | t◁ = t▷}) ({b::list(num) | adjacent b◁  b▷})
               : M_{s·t²·ε,0} [{l::list(num) | l◁ = l▷}] {
match t with
 | 0       -> unit []
 | 1 + t'  -> let M curdb = DualQuery t' b        in
              let M e = expmech b (build_quality t' curdb in)     in
              let M new_qry = sampleN s e            in
              unit ((opt new_qry) :: curdb)
}
```

Fig. 9: Pseudo-code for DualQuery in HOARe^2.

database as a list of natural numbers; adjacent databases are lists of the same length differing in one element. We represent the output of the mechanism as a list of selected records, each encoded as a natural number.

The algorithm performs t steps, producing one record of the synthetic database in every round. For each round, we first build a *quality score*—a function from queries to real numbers—based on the previously produced records, using the auxiliary function build_quality. If we think of the current records as forming an approximate database, the quality score measures how poorly the approximation performs on each query. We then sample s queries using the exponential mechanism with this quality score; queries with higher error are more likely to be selected. These queries are fed into an optimization function opt, which chooses the next record to add to the approximate database.

The only private operation we use in this example is the exponential mechanism. This is defined as a primitive implementing the mechanism presented in Theorem 2.4. The quality score we generate at each round i has sensitivity i, and so a draw form the exponential mechanism is $i\epsilon$-private. Since i is upper bounded by t and there are s samples per round, the privacy cost per round is bounded by $s{\cdot}t{\cdot}\epsilon$. With t rounds in total, the whole algorithm is $s{\cdot}t^2 \cdot \epsilon$-private. This guarantee is reflected in the type of DualQuery:

$$\{t :: \mathbb{N} \mid t_\triangleleft = t_\triangleright\} \to \{db :: \mathsf{list}(\mathsf{num}) \mid \mathsf{adjacent}(db_\triangleleft, db_\triangleright)\} \to \mathsf{M}_{s\cdot t^2\cdot\epsilon,0}[\{l :: \mathsf{list}(\mathsf{num}) \mid l_\triangleleft = l_\triangleright\}].$$

The type states that for two runs with adjacent databases, dualquery will return synthetic databases that are $s{\cdot}t^2 \cdot \epsilon$ apart, where t is the number of iterations and s is the number of samples used.

Further developments

The ideas of relational verification using HOARe^2 has been also used in the context of verifying incentive properties in auctions and games [Barthe et al. 2015].

6. RELATED WORK

Formal techniques have been applied to guarantee differential privacy using other computational models as well. Tschantz et al. [2011] consider a verification framework for interactive private programs, where the algorithm can receive new input and produce multiple outputs over a series of steps. Their approach is based on verifying the correct

use of differentially private primitives. Their programs are modeled by probabilistic I/O-automata, and they provide a proof technique based on probabilistic bisimulation. Their method is currently limited only to ϵ-differential privacy. Chatzikokolakis et al. [2014a] study a similar approach based on bisimulation in the context of process algebras. They also extend their approach to consider different metrics. Xu [2012] proposes techniques to ensure differential privacy in a distributed setting using a probabilistic process calculus.

Many other tools supporting development of differentially private programs have been designed without using formal methods. We describe some of them here. The first approach proposed to ensure ϵ-differential privacy was based on the idea of designing agents tracking at runtime the privacy budget consumption, and aborting the computation when the budget is exhausted. This approach was proposed by McSherry and implemented in PINQ [McSherry 2009] a set of encapsulation methods for LINQ—a SQL-like language embedded in C#. The idea behind the PINQ design is to use program annotations describing the amount of noise each component needs, and a runtime monitor to check that the total amount of noise respects the privacy bound ϵ. Airavat [Roy et al. 2010] combines an approach similar to PINQ with access control in a MapReduce framework. While PINQ is restricted to ϵ-differential privacy, Airavat can handle also approximate differential privacy using a runtime monitor for δ.

More recently several works have extended the approach of PINQ. Proserpio et al. [2014] extend PINQ with weighted datasets, which give natural descriptions of graph algorithms. Ebadi et al. [2015] extend PINQ with a formal model using ideas from provenance to track the data of individuals and to ensure a refined version of differential privacy named *personalized differential privacy*.

Mohan et al. [2012] propose a platform for private data analysis based on the sample-and-aggregate framework [Nissim et al. 2007] that optimizes utility for certain queries. One advantage of this approach is that it can be applied to programs considered as black-boxes, but it provides good accuracy guarantees only for a limited set of programs. Mir et al. [2013] develop a framework to release aggregate mobility data using differential privacy. Chatzikokolakis et al. [2014b] develop tools using differential privacy to protect the location of individuals in geo-location systems. Eigner et al. [2014] develop an architecture for distributed differential privacy using several mechanisms achieving optimal utility also in a distributed scenario. Erlingsson et al. [2014] develop a framework for achieving differential privacy based on the randomized response approach [Warner 1965]. Narayan et al. [2015] develop a system ensuring differential privacy and using zero knowledge proofs to guarantee the consistency of the program's output.

7. CONCLUSION AND FURTHER DIRECTIONS

Differential privacy is emerging as a gold standard for data privacy. Designing methods and tools for supporting the development of differentially private programs is crucial. We have summarized a range of approaches based on the use of logical and verification techniques to support the design of differentially private programs.

An appealing aspect of differential privacy is its support for composition schemes. Most of the techniques we have described in this article support the sequential and parallel composition described by Theorem 2.5 and Theorem 2.6, respectively. In the differential privacy literature there are other composition schemes that give better guarantees—similar accuracy but with less privacy cost. An important example is the "advanced composition" theorem from Dwork et al. [2010], which permits trading off ϵ with δ. Informally, the theorem states that if we run n programs, $P_1(D), \ldots, P_n(D)$, on the same dataset D, and if each of them is (ϵ, δ)-differentially private, then the resulting

program is roughly $(\sqrt{n}\epsilon, n\delta)$-differentially private. None of the methods we presented here support it yet.

The approaches we presented have focused so far on developing methods and tools to ensure the privacy bound neglecting the accuracy guarantee. However, guaranteeing a good accuracy for programs is fundamental in practice. Accuracy statements usually take the form $\Pr[|P(x) - x| \geq \alpha] \leq \beta$, expressing the fact that the program $P(x)$ returns a result at distance greater than α from the non-noised value x with probability less than β. An example is Fact 2.1. These accuracy statements are natural requirements for probabilistic computations, and they appear in many practical situations like energy-efficient computation, audio-video streaming programming, etc. Interestingly, the proofs of these statements pose important challenges to verification. Even when they are not too complex, they depend on a wide range of probabilistic tools like concentration or union bounds, properties of expectation, reasoning principles using independence or conditional independence of random variables, martingales, etc. These are all challenges that must be addressed in order to provide full support for differential privacy.

REFERENCES

Arthur Azevedo de Amorim, Marco Gaboardi, Emilio Jesús Gallego Arias, and Justin Hsu. 2014. Really Natural Linear Indexed Type Checking. In *IFL 2014*. DOI: http://dx.doi.org/10.1145/2746325.2746335

Gilles Barthe, George Danezis, Benjamin Grégoire, César Kunz, and Santiago Zanella Béguelin. 2013. Verified Computational Differential Privacy with Applications to Smart Metering. In *CSF*. 287–301. DOI: http://dx.doi.org/10.1109/CSF.2013.26

Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. 2015. Relational Reasoning via Probabilistic Coupling. In *LPAR 2015*. DOI: http://dx.doi.org/10.1007/978-3-662-48899-7_27

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, César Kunz, and Pierre-Yves Strub. 2014. Proving Differential Privacy in Hoare Logic. In *CSF'14*. http://dx.doi.org/10.1109/CSF.2014.36

Gilles Barthe, Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Pierre-Yves Strub. 2015. Higher-Order Approximate Relational Refinement Types for Mechanism Design and Differential Privacy. In *POPL 2015*. http://doi.acm.org/10.1145/2676726.2677000

Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella-Béguelin. 2012. Probabilistic Relational Reasoning for Differential Privacy. In *POPL'12*. http://doi.acm.org/10.1145/2103656.2103670

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *POPL 2014*. http://doi.acm.org/10.1145/964001.964003

T.-H. Hubert Chan, Elaine Shi, and Dawn Song. 2011. Private and continual release of statistics. *ACM Transactions on Information and System Security* 14, 3 (2011), 26. http://eprint.iacr.org/2010/076.pdf

Konstantinos Chatzikokolakis, Daniel Gebler, Catuscia Palamidessi, and Lili Xu. 2014a. Generalized Bisimulation Metrics. In *CONCUR 2014*. DOI: http://dx.doi.org/10.1007/978-3-662-44584-6_4

Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Marco Stronati. 2014b. A Predictive Differentially-Private Mechanism for Mobility Traces. In *PETS 2014*. DOI: http://dx.doi.org/10.1007/978-3-319-08506-7_2

Loris D'Antoni, Marco Gaboardi, Emilio Jesús Gallego Arias, Andreas Haeberlen, and Benjamin C. Pierce. 2013. Sensitivity Analysis using Type-Based Constraints. In *FPCDSL 2013*. http://doi.acm.org/10.1145/2505351.2505353

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating noise to sensitivity in private data analysis. In *TCC'06*. http://dx.doi.org/10.1007/11681878_14

Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science* 9, 3-4 (2014), 211–407. DOI: http://dx.doi.org/10.1561/0400000042

Cynthia Dwork, Guy N. Rothblum, and Salil P. Vadhan. 2010. Boosting and Differential Privacy. In *FOCS 2010*. IEEE. http://dx.doi.org/10.1109/FOCS.2010.12

Hamid Ebadi, David Sands, and Gerardo Schneider. 2015. Differential Privacy: Now it's Getting Personal. In *POPL 2015*. 69–81. http://dl.acm.org/citation.cfm?id=2676726

Fabienne Eigner and Matteo Maffei. 2013. Differential Privacy by Typing in Security Protocols. In *CSF'13*. http://dx.doi.org/10.1109/CSF.2013.25

Fabienne Eigner, Matteo Maffei, Ivan Pryvalov, Francesca Pampaloni, and Aniket Kate. 2014. Differentially private data aggregation with optimal utility. In *ACSAC 2014*. DOI:http://dx.doi.org/10.1145/2664243.2664263

Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *CCS'14*. DOI:http://dx.doi.org/10.1145/2660267.2660348

Marco Gaboardi, Emilio Jesús Gallego Arias, Justin Hsu, Aaron Roth, and Zhiwei Steven Wu. 2014. Dual Query: Practical Private Query Release for High Dimensional Data. In *ICML'14*. http://jmlr.org/proceedings/papers/v32/gaboardi14.html

Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Narayan, and Benjamin C Pierce. 2013. Linear dependent types for differential privacy. In *POPL 2013*. http://dl.acm.org/citation.cfm?id=2429113

Andreas Haeberlen, Benjamin C. Pierce, and Arjun Narayan. 2011. Differential Privacy Under Fire. In *USENIX Security 2011*. http://static.usenix.org/events/sec11/tech/full_papers/Haeberlen.pdf

Frank McSherry. 2009. Privacy integrated queries. In *SIGMOD'09*. http://doi.acm.org/10.1145/1559845.1559850

Frank McSherry and Ratul Mahajan. 2010. Differentially-private network trace analysis. In *SIGCOMM'10*. http://doi.acm.org/10.1145/1851182.1851199

Frank McSherry and Kunal Talwar. 2007. Mechanism Design via Differential Privacy. In *FOCS'07*. 94–103. http://doi.ieeecomputersociety.org/10.1109/FOCS.2007.41

Darakhshan J. Mir, Sibren Isaacman, Ramón Cáceres, Margaret Martonosi, and Rebecca N. Wright. 2013. DP-WHERE: Differentially private modeling of human mobility. In *Big Data '13*. 580–588. DOI:http://dx.doi.org/10.1109/BigData.2013.6691626

Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil P. Vadhan. 2009. Computational Differential Privacy. In *CRYPTO 2009*. DOI:http://dx.doi.org/10.1007/978-3-642-03356-8_8

P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler. 2012. GUPT: privacy preserving data analysis made easy. In *SIGMOD 2012*. http://doi.acm.org/10.1145/2213836.2213876

Arjun Narayan, Ariel Feldman, Antonis Papadimitriou, and Andreas Haeberlen. 2015. Verifiable Differential Privacy. In *EuroSys 2015*. DOI:http://dx.doi.org/10.1145/2741948.2741978

Arvind Narayanan and Vitaly Shmatikov. 2008. Robust De-anonymization of Large Sparse Datasets. In *S&P'08*. IEEE. DOI:http://dx.doi.org/10.1109/SP.2008.33

Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. 2007. Smooth sensitivity and sampling in private data analysis. In *STOC 2007*. DOI:http://dx.doi.org/10.1145/1250790.1250803

Davide Proserpio, Sharon Goldberg, and Frank McSherry. 2014. Calibrating Data to Sensitivity in Private Data Analysis. *PVLDB* 7, 8 (2014), 637–648. http://www.vldb.org/pvldb/vol7/p637-proserpio.pdf

Jason Reed and Benjamin C. Pierce. 2010. Distance Makes the Types Grow Stronger: A Calculus for Differential Privacy. In *ICFP'10*. http://dl.acm.org/citation.cfm?id=1863568

Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. 2010. Airavat: Security and Privacy for MapReduce. In *NSDI 2010*. http://www.usenix.org/events/nsdi10/tech/full_papers/roy.pdf

Michael Carl Tschantz, Dilsun Kaynar, and Anupam Datta. 2011. Formal Verification of Differential Privacy for Interactive Systems (Extended Abstract). *ENTCS* 276, 0 (2011), 61–79. DOI:http://dx.doi.org/10.1016/j.entcs.2011.09.015

Stanley L. Warner. 1965. Randomized Response: A Survey Technique for Eliminating Evasive Answer Bias. *J. Amer. Statist. Assoc.* 60, 309 (1965), 63–69. http://www.jstor.org/stable/2283137

L. Xu. 2012. Modular Reasoning about Differential Privacy in a Probabilistic Process Calculus. In *TGC 2013*. http://dx.doi.org/10.1007/978-3-642-41157-1_13