Differential Privacy: A Step Towards Protecting Our Data

Ryan Sowa
McGill University, Montréal, Canada
ryan.sowa@mail.mcgill.ca

Abstract. From data stored by large-scale agencies to data stored for scientific research to data stored in our cellphones, data is and will continue to be ubiquitous in our everyday lives. The privacy and protection of our data is, therefore, compulsory. One way to protect our data from breaches is by ensuring that every computation on sensitive data satisfies *differential privacy*, a rigorous framework for stating and enforcing privacy guarantees [1]. This phenomenon has given rise to tools which aim to check whether or not a program is differentially private. Many of the existing formal methods for differential privacy necessitate a a steep learning curve on the part of the programmer and can be quite tedious. In this paper, I will discuss and compare different tools that ensure differential privacy. I will also present a simple imperative language for verifying differentially-private programs which draws a balance between expressive power and usability.

1 Introduction

Over the years, several tools for achieving differential privacy have been introduced, each with its advantages and disadvantages. I will first define the notion of differential privacy and will go on to discuss various differential privacy tools analyzed by Barthe et al. [1], namely the *Fuzz* Approach and the *CertiPriv* Approach. I will elaborate upon the related syntax and type assignment rules as well as offer concrete examples to further my explanation. Finally, I will analyze *LightDP*, "a new imperative language for verifying sophisticated privacy-preserving algorithms [5]." The goal of *LightDP* is to "to minimize the burden on the programmer while retaining most of the capabilities of the state-of-the-art." To complete a thorough analysis, I will analyze the language syntax and typing rules for *LightDP*. Then, I will showcase the utility of *LightDP* by detailing a hands-on experiment I conducted with the tool. I will discuss my results in detail.

2 Differential Privacy Tools

2.1 Differential Privacy

According to Dwork, McSherry, Nissim, and Smith (2006, cited by [5]), differential privacy is "a set of restrictions on their probabilistic behavior that provably limit the ability of attackers to infer individual-level sensitive information." Dwork et. al define the notion

$$\tau, \sigma ::= \mathbb{R} \mid !_r\sigma \multimap \tau \mid \sigma \oplus \tau \mid \mathrm{M}\ \sigma$$

$$\begin{aligned} e ::= {} & x \mid \mathrm{r} \mid \mathrm{f} \mid \lambda x :!_r\sigma.e \mid e\ e \mid \textbf{fix}\ g.x.e \mid \textbf{inj}_i\ e \mid \textbf{case}\ e\ \textbf{of}\ x \to e\ \textbf{or}\ y \to e \\ & \textbf{let M}\ x = e\ \textbf{in}\ e \mid \textbf{unit}\ e \end{aligned}$$

$$\Gamma ::= \emptyset \mid \Gamma, x :!_r\sigma$$

Figure 1: *Fuzz* Syntax Rules

of differential privacy more formally below:

Definition 2.1.1 (Differential privacy: A randomized program $P : dB \to \mathcal{R}$ is (ϵ, δ)-differentially private for $\epsilon, \delta \geq 0$ if for every two databases, $D, D' \in dB$, differing in one row and for any subset $S \subseteq R$ of the outputs, we have:

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

.

Here, ϵ and δ represent key information: ϵ represents the *privacy cost* (i.e., the cost an individual's data will be exposed) and δ represents the trade-off between privacy and utility (it is typically 0).

2.2 The *Fuzz* Approach

One of the ways to ensure differential privacy is the *Fuzz Approach* [4]. The *Fuzz Approach* aims to ensure that programs are ϵ-differentially private (however, not (ϵ, δ)-differentially private) [3]. *Fuzz* is designed to use a type-checking procedure to ensure that well-typed programs of a particular type are ϵ-differentially private. To better explain how the *Fuzz Approach* works, we introduce the *Fuzz* syntax rules and typing rules [1]as given by Figures 1 and 2, respectively.

We also define the notion of *c-sensitivity* as follows:

Definition 2.2.1: Let $P : A \to B$ be a program and d_A and d_B be metrics on A and B, respectively. Then, P is c-sensitive for $c \in \mathbb{R}_{>0}$ if, for all $x, y \in A$, we have

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

Finally, we state the main theorem for differential privacy in *Fuzz* :

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

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

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

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

Figure 2: *Fuzz* Typing Rules

Theorem 2.2.1: Let $M\ \mathcal{R}$ represent the monadic type of discrete probability distributions over the output type $\mathcal{R}$, $\multimap$ represent the space of 1-sensitive functions from $!_\epsilon dB$ to $M\ \mathcal{R}$, and the type $!_\epsilon dB$ represent the type of databases whose metric is that of dB multiplied by ϵ. If

$$\vdash e :!_\epsilon dB \multimap M\ \mathcal{R}$$

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

As a final note, *Fuzz* only provides useful analyses when program sensitivities are statistically bounded - not when sensitivity depends on program inputs. This can be a problem when the total privacy level of a program depends on the number of iterations of a program. As a result of this shortcoming, tools like *DFuzz*, which allow dependent typing, have been introduced.

2.3 Example of *Fuzz*

To demonstrate how *Fuzz* works, we refer to the example given by Barthe et al [1]. Suppose that we want to determine how many patients at a hospital are over the age of 40. We are given the following functions:

over_40: Determines if an individual is over the age of 40

size: Outputs how many rows a database has

filter: Outputs how many rows satisfy a predicate

.

With these functions, we can formulate the following differentially-private program to address the query:

$$\lambda d :!_\epsilon dB.add\_noise(size(filter\ over\_40\ d)) :!_\epsilon dB \multimap M\mathcal{R}$$

.

2.4 The *CertiPriv* Approach

Another way to ensure differential privacy is the *CertiPriv* Approach [2]. The *CertiPriv* Approach aims to use *relational Hoare logic* to verify differential privacy [1].

Traditional Hoare Logic can be used to reason about a program using logical predicates as pre-conditions and post-conditions. In particular, if ϕ is a pre-condition and ψ is a post-condition, we can reason about a program c by using a judgment of the following form:

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

. This judgment expresses that given a memory m satisfying ϕ, c will produce a memory m' satisfying ψ.

This logic can be extended to two commands, c_1 and c_2:

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

. This judgment expresses that given two memories m_1 and m_2 satisfying the precondition Ψ, the commands c_1 and c_2 will produce a memory m_1' and m_2' satisfying the postcondition Φ.

The relational Hoare logic proposed by Barthe et al. [1] for reasoning about differential privacy was called *apRHL*. In particular, *apRHL* expresses judgments on `pWhile` commands c defined by the grammar:

$$c ::= \text{skip} \mid c; c \mid x \leftarrow e \mid x \overset{\$}{\leftarrow} \text{Lap}_\epsilon(e) \mid x \overset{\$}{\leftarrow} \text{Exp}_\epsilon(e, e) \mid \text{if } e \text{ then } c \text{ else } c$$

$$\text{while } e \text{ do c}$$

. The main theorem for differential privacy in *apRHL* is then given by:

Theorem 2.4.1: Let c be a ***`pWhile`*** *command and $c_\triangleleft$ and $c_\triangleright$ be two copies of c resulting from renaming of variables in c. If*

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

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

```
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;
```

Figure 3: Transformed Partial Sums Program

2.5 Example of *CertiPriv*:

To demonstrate how *CertiPriv* works, we refer to the example given by Barthe et al. [1]. Given a list of length n, we seek to create a differentially private algorithm for computing partial sums over lists of integer values taken from a bounded interval. The *CertiPriv Approach* aims to achieve a private approximation of computing these partial sums. The two approaches to achieving this goal are input perturbation (computing all partial sums over a "noised" list) and output perturbation (adding Laplace noise to each partial sum). However, there are upsides and downsides to both of these approaches. Namely, input perturbation offers poor privacy and good accuracy, while output perturbation offers poor accuracy but good privacy. *CertiPriv* combines both of these approaches to achieve a reasonable trade-off between privacy and accuracy. The transformed program is shown in Figure 3.

3 *LightDP*

3.1 Abstract

Due to the necessity to secure our data and prevent breaches, many algorithms used to ensure differential privacy such as the *Fuzz* Approach and the *CertiPriv* Approach were invented. However, not all of these algorithms were correct (i.e., they did not accurately ensure differential privacy), and a formal measure to prove whether an algorithm was differentially private was required. Many of these differential privacy-checking algorithms could verify sophisticated algorithms; however, they required extensive knowledge on the part of the programmer. *LightDP*, a simple imperative language, was invented to address this knowledge gap [5]. The goal of *LightDP* was to help develop provably privacy-preserving algorithms while also striking a better balance between expressive power and usability.

Reals	$r \in \mathbb{R}$
Booleans	$b \in \{\texttt{true}, \texttt{false}\}$
Vars	$x \in Var$
Rand Vars	$n \in H$
Linear Ops	$\oplus ::= + \mid -$
Other Ops	$\otimes ::= \times \mid /$
Comparators	$\odot ::= < \mid > \mid = \mid \leq \mid \geq$
Rand Exps	$g ::= \texttt{Lap}\ r$
Expression	$e ::= r \mid b \mid x \mid \eta \mid e_1 \oplus e_2 \mid e_1 \otimes e_2 \mid e_1 \odot_2 \mid$ $\neg e \mid e_1 :: e_2 \mid e_1[e_2] \mid e_1 ? e_2 : e_3$
Commands	$c ::= \texttt{skip} \mid x := e \mid \eta := g \mid c_1; c_2 \mid \texttt{return}\ e \mid$ $\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \mid \texttt{while}\ e\ \texttt{do}\ c$
Distances	$d ::= r \mid x \mid \eta \mid d_1 \oplus d_2 \mid d_1 \otimes d_2 \mid d_1 \odot d_2 ? d_3 : d_4$
Types	$\tau ::= \texttt{num}_d \mid \texttt{num}_* \mid \texttt{bool} \mid \texttt{list}\ \tau \mid \tau_1 \to \tau_2$

Figure 4: Syntax for *LightDP*

3.2 Typing Rules and Syntax

The syntax for *LightDP* is given in Figure 4. Based on this syntax, we can conclude that *LightDP* behaves like a standard imperative language with the exception of its use of random expressions, list operations, types with distances, and star types.

A subset of the typing rules for expressions for *LightDP* is shown in Figure 5. *LightDP*'s dependent type system aims to capture exact differences of a variable's values in two executions under adjacent databases. This typing system verifies that these differences are bounded as an invariant during the execution using type annotations. In verifying that this invariant is always maintained, the typing system ensures that the two related executions will always produce the same output. In addition, the type annotations reduce the annotation burden generally required by the programmer when verifying a program for differential privacy, thus improving usability.

3.3 Differential Privacy in *LightDP*

If type-checking succeeds, *LightDP* transforms the program it wants to verify into a non-probabilistic program where privacy cost is explicitly calculated by a new variable, v_ϵ. Then, the fundamental soundness theorem for *LightDP* informally states if v_ϵ is always bounded by some constant ϵ' in the transformed program, then the original program is ϵ'-differentially private.

$$\frac{}{\Gamma \vdash r : \texttt{num}_0}\text{T-NUM} \qquad \frac{}{\Gamma \vdash b : \texttt{bool}}\text{T-BOOLEAN}$$

$$\frac{}{\Gamma, x : \beta_d \vdash x : \beta_d}\text{T-VAR} \qquad \frac{}{\Gamma, x : \beta_* \vdash x : B_{\hat{x}}}\text{T-VARSTAR}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num}_{d_1} \quad \Gamma \vdash e_2 : \texttt{num}_{d_2}}{\Gamma \vdash e_1 \oplus e_2 : \texttt{num}_{d_1 \oplus d_2}}\text{T-OPLUS}$$

$$\frac{\Gamma \vdash e_1 : \texttt{num}_0 \quad \Gamma \vdash e_2 : \texttt{num}_0}{\Gamma \vdash e_1 \otimes e_2 : \texttt{num}_0}\text{T-OTIMES}$$

Figure 5: Typing Rules for *LightDP*. For simplicity, we just show a subset of the typing rules for expressions.

3.4 Example of *LightDP*

We now demonstrate how *LightDP* works using the following example. Given a list of queries, q representing trials counting the amount of individuals with an infection, we seek to calculate whether some query, $q[i]$, exceeds a threshold, T. The Sparse Vector method (the program on the top left in Figure 6) aims for a differentially private program to accomplish this. To verify if this program is differentially private, the program on the right in Figure 6 is constructed. In this transformed program, privacy cost is explicitly calculated via a new variable, v_ϵ and probabilistic instructions are replaced by a new nondeterministic instruction, `havoc η`. We can now conclude that if v_ϵ is always bounded by some ϵ' in the transformed program, then the original program is ϵ'-differentially private.

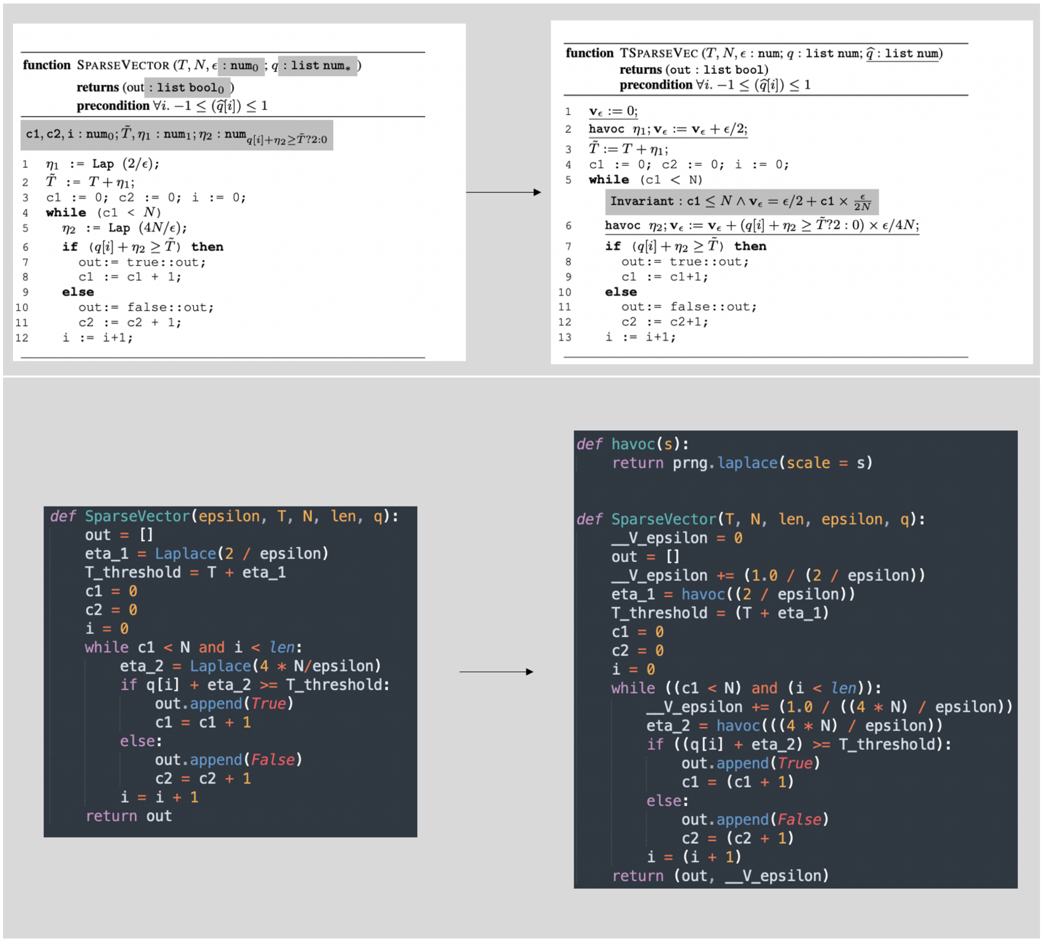
4 Methods

My goal was to verify a differentially-private program using *LightDP*. Specifically, I aimed to replicate the transformation of the `Sparse Vector` program discussed above. In order to accomplish this, I made use of a repository created by Yuxin Wang which presented a tool for transforming *LightDP* programs into Python programs along with differential-privacy proofs. This repository can be found at https://github.com/yxwangcs/lightdp. I ran this tool on my computer, hoping to be able to produce the same results obtained by Zhang and Kifer [5] in transforming the `Sparse Vector` program using *LightDP*.

5 Results

As shown in Figure 6, the results obtained by running the tool on my computer were very similar to those produced by Zhang and Kifer [5]. More specifically, the results revealed several differences between the original and transformed Python programs:

- A `havoc` function was defined and used in `SparseVector`



```
function SparseVector ($T, N, \epsilon$ : num$_0$ ; $q$ : list num$_*$ )
        returns (out : list bool$_0$ )
        precondition $\forall i.\ -1 \leq (\widehat{q}[i]) \leq 1$
c1, c2, i : num$_0$; $\tilde{T}, \eta_1$ : num$_1$; $\eta_2$ : num$_{q[i]+\eta_2 \geq \tilde{T} ? 2:0}$
$\eta_1$ := Lap $(2/\epsilon)$;
$\tilde{T}$ := $T + \eta_1$;
c1 := 0; c2 := 0; i := 0;
while (c1 < $N$)
  $\eta_2$ := Lap $(4N/\epsilon)$;
  if ($q[i] + \eta_2 \geq \tilde{T}$) then
    out:= true::out;
    c1 := c1 + 1;
  else
    out:= false::out;
    c2 := c2 + 1;
  i := i+1;
```

```
function TSparseVec ($T, N, \epsilon$ : num; $q$ : list num; $\widehat{q}$ : list num)
        returns (out : list bool)
        precondition $\forall i.\ -1 \leq (\widehat{q}[i]) \leq 1$
$\mathbf{v}_\epsilon := 0$;
havoc $\eta_1$; $\mathbf{v}_\epsilon := \mathbf{v}_\epsilon + \epsilon/2$;
$\tilde{T} := T + \eta_1$;
c1 := 0; c2 := 0; i := 0;
while (c1 < N)
  Invariant : c1 $\leq N \wedge \mathbf{v}_\epsilon = \epsilon/2 +$ c1 $\times \frac{\epsilon}{2N}$
  havoc $\eta_2$; $\mathbf{v}_\epsilon := \mathbf{v}_\epsilon + (q[i] + \eta_2 \geq \tilde{T} ? 2 : 0) \times \epsilon/4N$;
  if ($q[i] + \eta_2 \geq \tilde{T}$) then
    out:= true::out;
    c1 := c1+1;
  else
    out:= false::out;
    c2 := c2+1;
  i := i+1;
```

```python
def SparseVector(epsilon, T, N, len, q):
    out = []
    eta_1 = Laplace(2 / epsilon)
    T_threshold = T + eta_1
    c1 = 0
    c2 = 0
    i = 0
    while c1 < N and i < len:
        eta_2 = Laplace(4 * N/epsilon)
        if q[i] + eta_2 >= T_threshold:
            out.append(True)
            c1 = c1 + 1
        else:
            out.append(False)
            c2 = c2 + 1
        i = i + 1
    return out
```

```python
def havoc(s):
    return prng.laplace(scale = s)


def SparseVector(T, N, len, epsilon, q):
    __V_epsilon = 0
    out = []
    __V_epsilon += (1.0 / (2 / epsilon))
    eta_1 = havoc((2 / epsilon))
    T_threshold = (T + eta_1)
    c1 = 0
    c2 = 0
    i = 0
    while ((c1 < N) and (i < len)):
        __V_epsilon += (1.0 / ((4 * N) / epsilon))
        eta_2 = havoc(((4 * N) / epsilon))
        if ((q[i] + eta_2) >= T_threshold):
            out.append(True)
            c1 = (c1 + 1)
        else:
            out.append(False)
            c2 = (c2 + 1)
        i = (i + 1)
    return (out, __V_epsilon)
```

Figure 6: Transforming the `Sparse Vector` program using *LightDP*. The top diagram shows the abstract transformation of a program by *LightDP* as presented in Zhang and Kifer [5]. The bottom diagram shows the transformation after running Yuxin Wang's tool for transforming *LightDP* programs into Python programs.

- A new variable `__V_epsilon` was defined which kept track of privacy costs
- `__V_epsilon` is returned by `SparseVector`

. Therefore, based on how *LightDP* works, we can conclude that the original program is ϵ'-differentially private if `__V_epsilon` is always bounded by some ϵ' in the transformed program.

In terms of usability, this tool proved to be very simple to run and produce clear results. This would make it reasonably easy for someone with very little knowledge of differential privacy proofs to verify a differentially-private program.

6 Conclusion

In conclusion, the security of our data is fundamental as everyone, from individuals to large-scale corporations, uses sensitive data. Differential privacy is crucial to the protection of this data, and there exists several approaches to ensuring differential privacy. I mainly analyzed two approaches: the *Fuzz* Approach and the *CertiPriv* Approach. The *Fuzz* Approach uses type-checking procedures to ensure that well-typed programs of a particular type are ϵ-differentially private, while the *CertiPriv* Approach uses Hoare Logic to reason about programs. Finally, *LightDP* is a simple, effective tool to develop provably privacy-preserving algorithms. I conducted a hands-on experiment with *LightDP* by using a tool presented by Yuxin Wang for transforming *LightDP* programs into Python programs along with differential-privacy proofs. I ran this tool on the `Sparse Vector` program and was able to obtain similar results to those obtained by Zhang and Kifer. After conducting this experiment, I came to the conclusion that it would be relatively straightforward for someone with little knowledge of differential privacy proofs to verify a differentially-private program with *LightDP*.

References

[1] Gilles Barthe, Marco Gaboardi, Justin Hsu, and Benjamin Pierce. Programming language techniques for differential privacy. *ACM SIGLOG News*, 3(1):34–53, feb 2016.

[2] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, page 97–110, New York, NY, USA, 2012. Association for Computing Machinery.

[3] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In Shai Halevi and Tal Rabin, editors, *Theory of Cryptography*, pages 265–284, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[4] Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, page 157–168, New York, NY, USA, 2010. Association for Computing Machinery.

[5] Danfeng Zhang and Daniel Kifer. Lightdp: Towards automating differential privacy proofs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, page 888–901, New York, NY, USA, 2017. Association for Computing Machinery.