15-316: Software Foundations of Security and Privacy **Spring 2017**

Lecture 23: Language-Based Techniques for Differential Privacy

Lecturer: Jean Yang

In the last couple of weeks, you have learned about using *statistical privacy* to allow databases to release the results of queries over sensitive data. An open problem in differential privacy is the challenge in proving programs to be differentially private. Early methods have focused on having humans verify the differential privacy of specific algorithms. While this certainly ensures differential privacy, it does not scale well to developing lots and lots of differentially privacy algorithms!

As we have been learning in this class, program analysis can come in handy if we want to develop automated analyses of programs, and language design can give us the property that all programs in a given language are differentially private. For the systems we consider today, the threat model is as follows:

- The database an associated query system are hosted on a private, secure machine. The adversary does not have physical access, and can only communicate wit h it via the network.
- The adversary submits arbitrary queries over the network, and the system executes each safe query and returns the answer over the network.
- The system maintains a privacy budget for the database, and refuses to answer any more queries once the privacy budget is exhausted.

This lecture is based on papers by Haeberlen, Pierce, and Narayan [1], and Reed and Pierce [2].

23.1 Review of Differential Privacy

Intuitively, given a medical database the query *"How many patients at this hospital are over the age of 25?"* is intuitively "almost safe." It is safe because it aggregates many individuals' information together, making it difficult to deduce information about any individual, but it is *almost* safe because if an adversary knows the age of every patient except Matt Fredrikson, then they can deduce information about Matt's age. In contrast, the query *"How many patients named Matt Fredrikson are over the age of 25"* remains problematic, because such a query cannot be usefully privatized: adding enough noise to obscure the contribution of Matt's age makes it so that there will be no useful signal left.

Discussion question. What kinds of queries can be made statistically private, and what can't?

As a refresher, a randomized function is differentially private if arbitrary changes to a single individual's row, keeping other rows constant, result in only statistically insignificant changes to the function's output distribution, making it so that any individual's presence in the database has a statistically negligible effect. More formally, differential privacy is parameterized by a real number ϵ corresponding to the strength of the privacy guarantee, where smaller ϵ's correspond to greater privacy. Two databases b and b' are *similar*, or $b \sim b'$, if they differ in only one row, and a randomized function $q : \texttt{db} \to \mathbb{R}$ is ϵ-*differentially private* if, for all possible set of outputs $S \subseteq \mathbb{R}$, and for all similar databases b, b', we have

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

In other words, changing the input database by one row results in at most a very small multiplicative difference (e^ϵ) in the probability of *any* outcome set S. Recall that functions must be random to be differentially private. Adding noise is enough to make a function random, and it is a standard way of making functions differentially private.

23.1.1 Compositionality and Budgets

There are two main properties of differential privacy:

1. *Compositionality*: composing a differentially private function with any other function that does not depend on the database yields a function that is also differential privacy. An important consequence is that no amount of postprocessing can lessen the differential privacy guarantee.
2. *Graceful degradation:* a pair of two ϵ-differentially private functions is always 2ϵ-differentially private together. This allows us to think of a fixed "privacy budget:" with a privacy budget of ϵ, we can independently answer k queries, where the i^{th} query is ϵ_i-differentially private and $\Sigma_i \epsilon_i \leq \epsilon$.

23.1.2 Function Sensitivity

The main idea in proving programs are differentially private involves bounding the *sensitivity* of queries to small changes in their inputs. The amount of noise required to make a deterministic query differentially private is proportional to its sensitivity. For example, the sensitivity of the age queries from before is 1: adding or removing one patient's records from the hospital database can change the true value of each query by at most 1. Surprisingly, we should add the *same amount of noise* to the queries *"How many patients at this hospital are over the age of 25?"* as *"How many patients named Matt Fredrikson are over the age of 25?"* While these functions have the same sensitivity, however, the usefulness of the results is different: for the first query, a margin of error of ± 100 is still useful, while this margin of error is useless for the second query. It is possible to make the second query more useful by scaling the answer up numerically, for instance *"Is Matt Fredrikson over 25? If yes, then 1000, else 0."* But now this scaled-up query now has a sensitivity of 1000! Function sensitivity will be an important concept in analyzing programs for differential privacy.

23.2 Dynamic Approaches

There are two well-known research languages, PINQ and Airavat, that expose a set of operations to the programmer, where each operation depletes the privacy budget (also set by the programmer) in a predefined way. The languages have the following model: programmers write *queries* that consist of one or more *mapping* operations processing individual database records, together with *reducing* code that combines the results of mapping operations without directly looking at the database. The system verifies that each query is ϵ_i-differentially private, and deducts ϵ_i from the total budget ϵ associated with the database. If ϵ is nonzero, the system returns the query result. Otherwise, the user has exhausted the budget. The system throws away the database, reports that the budget has been exceeded, and never answers any more queries. We call the mapping operations *microqueries* and the rest of the code the *macroquery.*

In Airavat, a query is a sequence of chained microqueries ("mappers"), and a selection from a fixed set of macroqueries ("reducers"). The reducers are part of the trusted code base, and the mappers are the only untrusted code. When a user submits a query, they must also declare the expected numerical range of the outputs, which amounts to stating its sensitivity, since the input is a single record of the database. The system enforces the declared sensitivity by clipping the actual output if it falls outside of the declared range.

Airavat uses the declared sensitivity to calculate how much noise must be added to the reducer's results to achieve ϵ-differential privacy.

PINQ allows programmers to write macroqueries in LINQ, a SQL-like declarative language. The queries can be embedded in otherwise unconstrained C# programs. Microqueries can be general C# computations.

23.3 Type-Based Analysis for Differential Privacy

Now we show that we can statically analyze queries for their differential privacy, and to figure out how much noise to add. On the surface, this may not seem *that* much more powerful than Airavat or PINQ, but the fact that we can now determine these properties statically opens the doors for all sorts of things, for instance verifying the differential privacy of more expressive algorithms. We can also use this technique to close timing channels. The language we now describe underlies a PINQ-like language supporting the constructs `map`, `split`, `count`, and `sum`.

23.3.1 Type System

The underlying type system is based on a notion of *function sensitivity:* well-typed programs not only can't go wrong, but they *can't go too far* on nearby inputs. Using a monad for random computations, Reed and Pierce show that the definition of ϵ-differential privacy falls out as a special case of this soundness. This core calculus for differential privacy is based on a *distance-aware* type system that uses *linear logic*, which treats assumptions as consumable resources. It turns out that the capability to sensitively depend on an input's value behaves like a resource. The other component of the calculus involves using a *monad* to internalize the operation of adding random noise to query results.

23.3.1.1 A Crash Course in Linear Logic

The logic we have seen so far can be seen as wasteful. We only care about the truth; we don't care how many of each truth we have! We will now look at a simple logic where how many we have of each truth matters. Suppose we have the following propositions:

$1	having a dollar
candy	candy bar
chips	pack of chips
drink	soft drink
gum	gum

In the kinds of logic we have seen before, we might right $\$1 \Rightarrow \mathit{candy}$. But in that logic, from A and $A \Rightarrow B$, which means that we could keep our money *and* eat our candy. Restricting weakening ($\dfrac{\Gamma \vdash \Sigma}{\Gamma, A \vdash \Sigma}$, $\dfrac{\Delta \vdash \Sigma}{\Delta \vdash A, \Sigma}$) and contraction ($\dfrac{\Gamma \vdash \Sigma}{\Gamma \vdash A, \Sigma}$, $\dfrac{\Gamma \vdash A, A, \Sigma}{\Gamma \vdash A, \Sigma}$) allows linear logic to rule out this kind of reasoning. We introduce a notion of *linear* implication $\$1 \multimap \mathit{candy}$ to express that we are allowed to transform $1 into *candy*. From $1 and this fact, we can conclude *candy*.

Here are the other connectives:

- Multiplicative conjunction ($A \otimes B$) denotes simultaneous occurrence of resources, to be used as the consumer directs. If you buy a stick of gum and a bottle of soft drink, then you are requesting *gum* $\otimes$ *drink*. From \$1 and \$1 $\to$ *candy*, we cannot conclude \$1 $\otimes$ *candy*. The constant 1 denotes the absence of any resource and is the unit of $\otimes$.

- Additive conjunction (A & B) represents alternative occurrence of resources, the choice of which is up to the consumer. We write \$1 $\multimap$ (*candy* & *chips* & *drink*) to denote that you can buy exactly one of a candy bar, a pack of chips, or a soft drink with \$1. Writing \$1 $\multimap$ (*candy* $\otimes$ *chips* $\otimes$ *drink*) would denote that we could buy all three for \$1. From \$1 $\multimap$ (*candy* & *chips* & *drink*) we can deduce \$3 $\multimap$ (*candy* $\otimes$ *chips* $\otimes$ *drink*), where \$3 := \$1 $\otimes$ \$1 $\otimes$ \$1. The unit $\top$ may be used as a "wastebasket" for irrelevant alternatives. For example, we can write \$3 $\multimap$ (*candy* & $\top$) to convey that three dollars will buy a candy bar and something else.
- Additive disjunction ($A \oplus B$) represents alternative occurrence of resources, where the machine controls which resource. The linear implication \$1 $\multimap$ (*candy* $\oplus$ *chips* $\oplus$ *drink*) describes a situation where we have a vending machine that might return either candy, chips, or a soft drink. The unit of $\oplus$ is 0, which represents a product that cannot be made.

Here is the rule for multiplicative conjunction ($\otimes$):

$$\frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}$$

Here are the rules for additive conjunction (&) and disjunction ($\oplus$):

$$\frac{\vdash \Gamma, A \quad \vdash \Gamma, B}{\vdash \Gamma, A \,\&\, B} \quad \frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B} \quad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}$$

23.3.2 Using Linear Logic for Sensitivity

The type system underlying Fuzz uses linear logic to propagate function sensitive. We begin by defining sensitivity for real-valued functions:

Definition 23.1 *A function $f : \mathbb{R} \to \mathbb{R}$ is said to be c-sensitive iff $d_{\mathbb{R}}(f(x), f(y)) \leq c \cdot d_{\mathbb{R}}(x, y)$ for all $x, y \in \mathbb{R}$.*

A special case of this definition is the case where $c = 1$. This is also called a *non-expansive* function, since it keeps distances between input points the same, or makes them smaller. Here are some examples of 1-sensitive functions:

$$f_1(x) = x \quad f_2(x) = -x \quad f_3(x) = x/2 \quad f_4(x) = |x| \quad f_5(x) = (x + |x|)/2$$

The function $f_6(x) = 2x$ is 6-sensitive, and the function $f_7(x) = x^2$ is not c-sensitive for any c. We have the following property:

Proposition 23.2 *Every function that is c-sensitive is also c'-sensitive for every $c' \geq c$.*

So far, we only have one type $\mathbb{R}$. We can now require that for every type τ in our system, there is a metric $d_\tau(x, y)$ for values $x, y \in \tau$ that tells us how far apart x and y are. This allows us to generalize the function of c-sensitivity to arbitrary types:

Definition 23.3 *A function $f : \tau_1 \to \tau_2$ is said to be c-sensitive iff $d_{\tau_2}(f(x), f(y)) \leq c \cdot d_{\tau_1}(x, y)$ for all $x, y \in \tau_1$.*

We now define a scaling operator $!_r\tau$:

$$d_{!_r\tau}(x, y) = r \cdot d_\tau(x, y)$$

This operator has the following property:

Definition 23.4 *A function f is a c-sensitive function in $\tau_1 \to \tau_2$ if and only if f is a 1-sensitive function in $!_c\tau_1 \to \tau_2$.*

In our setting an input of type $!_r\tau$ is analogous to a resource that can be used at most r times.

We can now use multiplicative conjunction to build up new metric-carrying types from existing ones, corresponding to combined distances. If τ_1 and τ_2 are associated with metrics d_{τ_1} and d_{τ_2}, then let $\tau_1 \otimes \tau_2$ be the types whose values are pairs (v_1, v_2) where $v_1 \in \tau_1$ and $v_2 \in \tau_2$. We define the distance between two pairs to be the sum of the distances between each pair of components:

$$d_{\tau_1 \otimes \tau_2}((v_1, v_2), (v_1', v_2')) = d_{\tau_1}(v_1, v_1') + d_{\tau_2}(v_2, v_2')$$

Using this operator we can describe arithmetic operations on real numbers. For instance, the following are 1-sensitive functions in $\mathbb{R} \otimes \mathbb{R} \to \mathbb{R}$:

$$f_8(x, y) = x + y \quad f_9(x, y) = x - y$$

The following are 1-sensitive functions in $\mathbb{R} \otimes \mathbb{R} \to \mathbb{R} \otimes \mathbb{R}$:

$$f_{10} = (x, y) \quad f_{11}(x, y) = (y, x) \quad f_{12}(x, y) = (x + y), 0) \quad cswp(x, y) = \begin{cases} (x, y) & \text{if } x < y \\ (y, x) & \text{otherwise} \end{cases}$$

The following functions are not 1-sensitive in $\mathbb{R} \otimes \mathbb{R} \to \mathbb{R} \otimes \mathbb{R}$:

$$f_{13}(x, y) = (x \cdot y, 0) \quad f_{14}(x, y) = (x, x)$$

Note that f_{14} is particularly interesting because we never risk multiplying x by more than 1, but the fact that x is used twice means that the variation of x is doubled in measurable variation of the output. The connection between the type system and linear logic captures this!

Another metric that is useful is taking the distance between pairs to be the maximum of the components, rather than the sum. The type $\tau_1 \ \&\ \tau_2$ consists of pairs $\langle v_1, v_2 \rangle$ with the metric

$$d_{\tau_1 \ \&\ \tau_2}(\langle v_1, v_2 \rangle, \langle v_1', v_2' \rangle) = \max(d_{\tau_1}(v_1, v_1'), d_{\tau_2}(v_2, v_2'))$$

Now, we can say $f_{15}(x, y) = (x, x)$ is a 1-sensitive function in $\mathbb{R} \otimes \mathbb{R} \to \mathbb{R} \ \&\ \mathbb{R}$. As we see, & lets us combine outputs to c-sensitive functions even if they depend on common inputs. We have the following property:

Proposition 23.5 *If $f : \tau \to \tau_1$ and $g : \tau \to \tau_2$ are c-sensitive, then $\lambda x.\langle f\ x, g\ x \rangle$ is a c-sensitive function in $\tau \to \tau_1 \ \&\ \tau_2$.*

Now we can talk about the set of functions itself. Let us define $\tau_1 \multimap \tau_2$ as the type whose values are 1-sensitive functions $f : \tau_1 \to \tau_2$. (Note that we established that having $!_r$ means that 1-sensitivity is enough to express c-sensitive functions for all c. The type of c-sensitive functions from τ_1 to τ_2 is just $!_c\tau_1 \multimap \tau_2$.) We define the metric for $\multimap$ as follows:

$$d_{\tau_1 \multimap \tau_2}(f, f') = \max_{x \in \tau_1} d_{\tau_2}(f(x), f(x'))$$

This is chosen to ensure that $\multimap$ and $\otimes$ have the expected currying and uncurrying behavior. In fact,

$$curry(f) = \lambda x.\lambda y.f(x,y)$$
$$uncurry(g) = \lambda(x,y).g\ x\ y$$

are 1-sensitive functions in $(\mathbb{R}\otimes\mathbb{R}\multimap\mathbb{R})\to(\mathbb{R}\multimap\mathbb{R}\multimap\mathbb{R})$ and $(\mathbb{R}\multimap\mathbb{R}\multimap\mathbb{R})\to(\mathbb{R}\otimes\mathbb{R}\multimap\mathbb{R})$, respectively.

Reed and Pierce provide formal definitions for this language, and prove a theorem that every well-typed expression $b :_c \texttt{db} \vdash e : \mathbb{R}$ is a c-sensitive function $\ \to \mathbb{R}$.

23.3.2.1 A Calculus for Differential Privacy

We can now apply this type system for expressing differentially private computations by using a monad for adding noise to c-sensitive functions. First, we can generalize our definition of differential privacy to functions, using the distance between inputs instead of the database similarity condition:

Definition 23.6 *A random function $q : \tau \to \sigma$ is ϵ-differentially private if for all $S \subseteq \sigma$, and for all $v, v' : \tau$, we have $Pr[q(v) \in S] \le e^{\epsilon d_\tau(v,v')} Pr[q(v') \in S]$.*

One way to achieve differential privacy is by adding *Laplace-distributed noise* to the result of f. We have the following property that the amount of noise required to make a c-sensitive function ϵ-private is c/ϵ:

Proposition 23.7 *Suppose $f : \texttt{db} \to \mathbb{R}$ is c-sensitive. Define the random function $q : \texttt{db} \to \mathbb{R}$ by $q = \lambda b.f(b) + N$, where N is a random variable distributed according to $\mathcal{L}_{c/\epsilon}$. Then q is ϵ-differentially private.*

We can use a *monad* to track the amount of noise to add to a computation. Monads are mathematical objects that help us model the chaining of computations in some context. We can define something as a monad if we can define the following two operations for values of any type: *return* for putting values into the monad, and *bind* for sequencing operations within the monad. We can model adding noise with extending the language with a monad of random computations. We add $\bigcirc\tau$, the type of random computations of τ. Expressions now include a monadic return **return** x, which deterministically yields x, as well as monadic sequencing: the expression **let** $\bigcirc = e$ **in** e' draws a sample x from the random computation e and then continues with the computation e'. Values now additionally return a probability distribution δ. The metric on probability distributions is defined to allow the type system to talk about differential privacy:

$$d_{\bigcirc\tau}(\delta_1,\delta_2) = \frac{1}{\epsilon}(\max_{x\in\tau} |\ln(\frac{\delta_1(x)}{\delta_2(x)})|)$$

The typing rules for the monad are as follows:

$$\text{I}\ \frac{\Gamma \vdash e : \tau}{\infty\Gamma \vdash \textbf{return}\ e : \bigcirc\tau} \qquad \text{E}\ \frac{\Delta \vdash e : \bigcirc\tau \quad \Gamma, x :_\infty \tau \vdash e' : \bigcirc\tau'}{\Delta + \Gamma \vdash \textbf{let}\ \bigcirc x = e\ \textbf{in}\ e' : \bigcirc\tau'}$$

The introduction rule multiples the context by infinity, because nearby inputs do not lead to nearby deterministic probability distributions: even if t and t' are close, **return** t still has a 100% chance of yielding t. The elimination rule adds together the influence Δ that e may have over the final output distribution to the influence Γ that e' has and gives variable x to e' without restriction. Once a differentially private query is made, the public result can be used in any way at all.

We add the following rules to our operational semantics:

$$\frac{e \hookrightarrow v}{\textbf{return } e \hookrightarrow (1, v)} \qquad \frac{e_1 \hookrightarrow (p_i, v_i)_{i \in I} \quad \forall i \in I.[v_i/x]e_2 \hookrightarrow (q_{ij}, w_{ij})_{j \in J_i}}{\textbf{let } \bigcirc x = e_1 \textbf{ in } e_2 \hookrightarrow (p_i q_{ij}, w_{ij})_{i \in I, j \in J_i}}$$

Here, **return** creates a trivial distribution that always yields v. Monadic sequencing considers all possible v_i that e could evaluate to, and then all values e' could evaluate to, assuming it received the sample v_i. The rule combines these two probabilities appropriately.

We have the following properties:

Lemma 23.8 *A* 1*-sensitive function* $\tau \to \bigcirc\sigma$ *is the same thing as an* ϵ*-differentially private random function* $\tau \to \sigma$*.*

Corollary 23.9 *The execution of any closed program* e *such that* $\vdash e :!_n\tau \multimap \bigcirc\sigma$ *is an* $(n\epsilon)$*-differentially private function from* τ *to* σ*.*

References

[1] A. Haeberlen, B. C. Pierce, and A. Narayan. Differential privacy under fire. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 33–33, Berkeley, CA, USA, 2011. USENIX Association.

[2] J. Reed and B. 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, pages 157–168, New York, NY, USA, 2010. ACM.