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

Lecture 20: Introduction to Data Privacy

Lecturer: Matt Fredrikson

20.1 Big Picture

We've spent some time discussing information flow controls, including a few techniques for relaxing information flow and understanding what consequences might follow from semantic underpinnings. In particular, we've shown that certain functions are probably fine to declassify in most situations, because an unauthorized user who wishes to learn a secret by way of such a function would need an unreasonble set of resources. However, even for simple functions like **match**, reaching this conclusion was a fairly involved process, and involved making a number of assumptions that may not always hold in practice. Let us recall some of the particularly troublesome assumptions that were necessary.

Deterministic vs. non-deterministic attackers. We assumed that the attacker making queries to the declassified function was deterministic. This was a reasonable assumption at the time, because we were adding the primitive to a deterministic language. In general, we may need to question such an assumption. What would happen if we were to make **match** available over a network interface, or as a system-wide API that could be called by arbitrary applications? Although the idea of a purely nondeterministic attacker may not correspond to a reality that we need to concern ourselves with, it isn't too much of a stretch to imagine an attacker who behaves probabilistically according to some distribution.

Distribution of secrets. We also chose not to consider the possibility that our secrets might be distributed in a way that makes "lucky" guesses occur with reasonable frequency. If we conceive of an attacker who is either very aggressive—i.e., one who is unwilling to make distributional assumptions so as to appeal to a wider range of scenarios—or who is ignorant of the fact that the secret may be fortuitously distributed, then this is a reasonable assumption. However, in many situations we must assume not only that the value we wish to protect is drawn from a somewhat skewed distribution, but that the attacker is aware of this fact and is willing to leverage it. Examples of sensitive data that fell in this category include users' passwords, gender, ethnicity, personal tastes, and almost anything else governed by natural law or human tendency.

At the same time, some of the most lucrative and appealing applications of computing are based on processing data collected from individuals, and releasing various aggregates to aid marketing, medical treatment, finance, and other forms of useful automation. While the semantic methods we have discussed so far give us a principled way of reasoning about the risk posed by such aggregate release of information, there are a few impediments to their broad application in such settings.

- They do not necessarily scale to the complexity of common data processing scenarios. Rather than reasoning from first principles about the risk posed by every instance of information release we would like to implement, it would be better to have a theory that abstracts from these details to tell us when such an operation is safe according to some policy.
- While the semantics of a program give us a way of assessing risk, they offer no guidance about what sort of risk is acceptable in various situations, or what sorts of risk are even avoidable.

- They do not by themselves suggest feasible or practical ways of mitigating this risk. Although we're free to declassify whatever we'd like in principle, we are left to our own ingenuity when it comes to figuring out how to safely release the information we need.

So far we've been focused on security, broadly defined. The set of theory and techniques grouped under the term *privacy* give us effective ways of addressing the concerns described above.

20.2 Setting: Statistical Database Model

For the purposes of our discussion, we will focus on a class of computations called the statistical database model. Intuitively, computations in this model allow external users access to a protected database via a limited set of **statistical queries**. Examples of such queries include counts, sums, averages, and measures of variance and covariance. A centralized process computes each query, and returns only the result to the user. In more detail, this model is composed of the following components.

Database. The database is the key element in this setup: users pose queries which are computed over the contents of the database. We don't require anything special from the database in terms of its structure or contents, and view it as a typical row-structured table containing information from arbitrary domains. We assume that it is of size $N \times M$, where each of the N rows corresponds to the data of a *single individual*, and each of the M columns contains exactly one *feature* (e.g., attribute) of an individual. We will use the symbol X to denote the database, and $x_1, \ldots, x_N$ to denote the individual rows.

Data Processor. The intermediary between the database and the users is called the *data processor*. The data processor is tasked with ensuring that access to the database only occurs through statistical queries, and that it does its job competently (i.e., we generally aren't concerned with system-level vulnerabilities that might lead to disclosure through other channels). Any policies or mitigation strategies that are to be applied towards protecting the database are implemented in the data processor. In general, the data processor will seek to implement policies that protect the information given by each individual in the database. We will use the symbol San to denote the data processor[1]

Users. The users of the system pose queries to the data processor, who runs them over the database. The goal of the users is to learn as much about the database as the data processor, and thus the set of queries for which they receive responses, will allow. Additionally, users may arrive with a certain set of *background knowledge*, and pose queries adaptively towards this goal. If we think of an "attacker" in this setting, then that attacker corresponds to a user. However, this goal is not necessarily malicious, as the users may simply benefit from accurate query results given by the processor. We will denote a prototypical user with the symbol A, as we will in general not have occassion to refer to multiple users at the same time.

Thus, the statistical database model presents a conflicting set of goals among the data processor and the user. The data processor is concerned with making sure that information about individuals in the database remains secret, while the users wish to learn accurate statistics about the data. The goal of our study will be to formalize these goals so that precise connections can be made between our assumptions and the degree to which these goals can be met. We will also look for concrete, practical techniques for achieving these goals in software.

[1] San comes from "sanitize" or "sanitizer", which is historically how the data privacy community has viewed the role of the data processor.

Name	Sex	Major	Class	SAT	GP
Allen	*Female*	*CS*	1980	600	3.4
Baker	*Female*	*EE*	1980	520	2.5
Cook	*Male*	*EE*	1978	630	3.5
Davis	*Female*	*CS*	1978	800	4.0
Evans	*Male*	*Bio*	1979	500	2.2

Figure 20.1: Database used in Example 20.1.

20.3 Models of Data Privacy

Now that we have the basic framework laid out, let's take a look at several potential ways of formalizing the goals of the data processor and users. Until we find a promising candidate, we won't worry too much about what specific techniques may be possible, and we'll instead focus on whether the high-level goals stand a chance of being realized in a satisfactory manner.

20.3.1 Exact Disclosure

One appealing approach is to simply return exact query results as users ask them. This is clearly optimal in terms of the users' goals, as they get exactly the information they asked for. What sort of privacy goal is reasonable for the data processor to adopt in this model? To gain an intuition, let's look at a relatively simple example of a statistical database.

Example 20.1 *Consider the database shown in Fig 20.2. Suppose that we create a statistical database that supports queries of the form* **count**(e)*, where e is a Boolean expression whose variables can range over* Sex, Major, Class, SAT, GP*.* **count** *queries behave exactly as one would expect: they return the number of rows in X that cause e to evaluate to* **true***. Users can submit as many queries of this form, and learn the exact result.*

Now suppose that A *from Example 20.1 has background knowledge that individual Cook is a* Male *majoring in* EE *from the class of* 1978*. Can* A *learn Cook's SAT score? Yes, and the attack is straightforward.*

1. A *sends query* $\textbf{count}(Sex = Male \wedge Major = EE \wedge Class = 1978)$.
2. *For each value v in the set* $\{310, 320, \ldots, 800\}$, A *sends query*

$$\textbf{count}(Sex = Male \wedge Major = EE \wedge Class = 1978 \wedge SAT = v)$$

3. A *takes Cook's SAT score to be the value which caused the latter query to return 1.*

Why does this attack work? First of all, there is only one male EE major from the class of 1978 in the database, so A initially got "lucky" as the first query returned 1. This tells A that when the correct guess for SAT is added to the initial query, the result will remain 1, and otherwise it will be 0.

It might be tempting to write this example off as a corner case, and conclude that exact disclosure is OK "most" of the time. For example, perhaps we can just impose the requirement that **count** queries aren't allowed over a set of variables that can uniquely identify an individual. However, we see that this doesn't improve the situation very much. Going back to the previous example, suppose that there were another individual added to the database: [Frank, *Male*, *EE*, 1978, 800, 3.9].

Name	Sex	Major	Class	SAT	GP
Allen	*Female*	*CS*	1980	600	3.4
Baker	*Female*	*EE*	1980	520	2.5
Cook	*Male*	*EE*	1978	630	3.5
Davis	*Female*	*CS*	1978	800	4.0
Evans	*Male*	*Bio*	1979	500	2.2
Frank	*Male*	*EE*	1978	800	3.9
Good	*Male*	*Bio*	1980	750	3.8

Figure 20.2: Database used in Example 20.2.

In this case, the initial **count** query would return 2 instead of 1. In the second phase of the attack, A would see the same result of 1 when querying $SAT = 630$ and $SAT = 800$. While this does not provide enough information to infer Cook's SAT score with perfect certainty, it does allow him to narrow it down to just two potential values. This in itself might be problematic, but if A also happens to know that Cook didn't have a perfect SAT, then exact information can again be recovered.

One potential approach to addressing this problem is to adopt a policy where the data processor will only return **count** results when they are in the range $[n, N - n]$[2]. The idea here is that **count** results close to the minimum or maximum values give A too much information about individual values, but perhaps results approximately near the middle of the range of possible values don't leak too much.

Example 20.2 *Suppose* A *knows that Evans is a male Biology major from the class of 1979, and again wishes to infer his SAT score. However, the data processor has now adopted a policy where* **count** *queries will only be returned if they evaluate to a number in the range* $[3, 4]$ *(i.e., the policy from above with* $n = 3$*). Thus, the query* $e \equiv Sex = Male \wedge Major = Bio \wedge Class = 1979$ *is not allowed, and* A *cannot learn that Evans is uniquely identified by this characteristic.*

Can A *still learn Evans' SAT score? Consider the following approach.*

1. *Let* e' *be the expression* $Sex = Female$*.* A *queries* $\mathbf{count}(e')$*, letting* r' *hold the result.*
2. A *sends queries* $\mathbf{count}(e' \vee (e \wedge SAT = v))$ *for all possible SAT scores, letting* r_v *hold the result.*
3. *Any* v *for which* $r_v - r' = 1$ *is a plausible SAT score for Evans.*

When this is over, only one v *is possible, revealing Evans' score.*

In the previous example, A padded each query with an additional "dummy" set that was sized appropriately for the queries to return successfully. By looking at the differences between the dummy set results by themselves, and the dummy set combined with Evans' characteristic, A was able to infer the desired information.

Looking back at the previous two examples, it seems that the exact query model is very difficult to reason about, even when the supported queries are simple in nature. Although in certain cases it may be possible to adopt a reasonable query policy that prevents A's feasible set from collapsing to a single point, doing so will require careful consideration of the particular context in which the approach is used. We'll move on, and consider stronger models in hopes of finding one that will apply more generally without extensive context-specific reasoning.

[2]Question: think about why we would want to restrict large results as well as small ones. What could A do to learn the same information from a large **count** result as a small one?

20.3.2 Absolute Privacy

Rather than focusing on the specifics of the release mechanism and focusing on utility, let's approach this problem from the other direction and think about what sort of privacy guarantee would be ideal to achieve. One simple idea is that *access to a database should not enable one to learn anything about an individual that could not be learned without access.* Is this goal attainable? Our better intuition would suggest that it isn't, because the point of accessing a database is to learn *something* about it, and this will be derived from the individual rows. However, to probe this question in more depth we'll need to formalize some of the concepts we've already discussed, and introduce a few more.

Data processor. So far, we've described the statistical database model as a form of interactive computation where users A pose queries to the data processor San, and receive answers in some form. To make our discussion more general, we won't pay any attention to the form of the queries, and we'll think of San as a function over the database X. In this model, users see the value $\mathsf{San}(X)$, and don't send it queries. Note that we haven't necessarily lost anything by doing this, because $\mathsf{San}(X)$ could simply return the result of evaluating all possible queries on X at once.

Background information. We said that users may have some kind of "background knowledge" that they can combine with information given by San to learn additional facts. In the previous section, we gave a few different examples of background information that could be problematic. Now, we'll generalize this concept by formalizing background information as the result of an **auxiliary information generator** G, which is a function of the database X. A receives the result of $G(X)$.

Privacy breach. This notion of privacy is very general in that the range of events that constitute a privacy breach is broad. Because of this, it doesn't make sense to focus on any particular type of breach (e.g., complete leakage of the secret, leaking confidence intervals, etc.), and instead we'll introduce the concept of a **breach decider**. For our purposes, the breach decider is just a function C that takes the database X and a **breach string** s as input, and returns 1 if s constitutes a breach of some individual's privacy in X. What constitutes a breach is left open—we assume it could be any computable function. For instance, we might define a breach function corresponding to leakage of SAT scores from the previous set of examples as follows:

$$C(X, s) = \begin{cases} 1 & \text{if } s = (i, v) \text{ and } X_i[SSN] = v \\ 0 & \text{otherwise} \end{cases}$$

However, we will assume that the breach is not "trivial", in the sense that the probability of producing a breach string by random guessing, without additional information about X, is low.

Users. Until now, we've assumed that the user A isn't necessarily malicious, but is interested in learning as much as possible about the database X. Now we'd like to know if our notion of absolute privacy is ever attainable, and we'll proceed by reasoning about whether and how often users are able to violate privacy. This is tantamount to assuming that users are always malicious, and more precisely in our model, whether they are able to produce breach strings that cause the breach decider C to evaluate to 1. Thus, we'll assume that A is a function taking the background information $G(X)$ and the information provided by the data processor $\mathsf{San}(X)$, and produce a string s with the intent of causing $C(X, s) = 1$. Additionally, we don't necessarily want to assume that users are deterministic, so we'll say that A is a randomized function.

Utility. Although we've chosen to focus first on finding the right definition of privacy, we can't forget about utility completely. Doing so would allow trivial instantiations of San, such as constant functions that

reveal no information about their inputs. We'll adopt a general notion of utility which requires that the information released by San should allow A to answer a series of yes/no questions about X. We formalize this by incorporating a **utility vector** w_X, which we can intuitively think of as containing the answers to the yes/no questions. We require that w_X should be efficiently computable from $\mathsf{San}(X)$, so that A can easily recover w_X. At the same time, we don't want w_X to be easily guessable without access to $\mathsf{San}(X)$, as this would defeat the whole purpose of providing access to the database to begin with. Thus, we'll assume that w_X is infeasible to compute without access to $\mathsf{San}(X)$.

Absolute privacy, refined. Now that we've got a better idea of how the various components in our setting fit together, let's return to our desired notion of privacy. We want it to be the case that whatever San releases doesn't give any additional information about the individual rows in X that would have been available otherwise. We've formalized this "additional information" as the set of breach strings s that cause $C(X, s) = 1$, so we roughly want the likelihood of the event $C(X, \mathsf{A}(\mathsf{San}(X), G(X))) = 1$ to be low. But this requirement is missing the "would have been available otherwise" part of our definition. How do we formalize this?

Before we can do so, we need to decide what we mean by "otherwise". Our focus is on understanding the privacy consequences of the data processor's release, $\mathsf{San}(X)$, and we're only concerned with the individual rows of X. The only other entities that could contain information about X are $D(X)$ and w_X. We've discussed our intuition for w_X, namely that it shouldn't be easy to compute without access to $\mathsf{San}(X)$; in other words, we view it as a function of $\mathsf{San}(X)$. But we make no such assumptions about $G(X)$, and in fact should probably assume that it's information that is available to anyone.

This suggests that we can formalize this idea of "would have been available otherwise" by way of a *simulator*, who attempts to behave exactly as A would, but without access to $\mathsf{San}(X)$. More precisely, our simulator is a function Sim that takes as input $G(X)$, and tried to produce a successful breach string. Because the simulator only sees $G(X)$, its behavior allows us to understand what the "baseline" risk of a privacy breach is even without access to $\mathsf{San}(X)$. With these concepts in mind, we can now define somewhat-formally what it means for San to protect the privacy of X.

Definition 20.3 *Fix a breach decider C. Then* San *preserves* absolute privacy *if for any user* A*, simulator* Sim*, database X, utility vector w_X, and background data $G(X)$, the following holds:*

$$\Pr[C(X, \mathsf{A}(\mathsf{San}(X), G(X))) = 1] - \Pr[C(X, \mathsf{Sim}(G(X))) = 1] \geq \Delta$$

Where the probabilities are taken over the random behavior of A *and* Sim*.*

Can we ever hope to satisfy this definition? Consider the following steps for the auxiliary information generator G and user A.

Algorithm for auxiliary information generator $G(X)$:

1. On receiving the database X, compute a breach string s.
2. Compute the utility vector w_X.
3. Let $k \leftarrow \mathsf{Hash}(w)$.
4. Return $\mathsf{Encrypt}_k(s)$.

Algorithm for user $\mathsf{A}(\mathsf{San}(X), G(X))$:

1. On receiving $\mathsf{San}(X)$, compute w_X.
2. Let $k \leftarrow \mathsf{Hash}(w)$.
3. Let $s \leftarrow \mathsf{Decrypt}_k(G(x))$.
4. Return s.

These choices of G and A work in tandem, so that the auxiliary information generator encrypts a breach string (which it can easily compute, having access to all of X) with a key derived from w_X. A, being able to efficiently compute w_X, then reconstructs this key and decrypts the breach string. In this way, as long as a breach string can be effeciently derived from X, the user will always succeed in breaching privacy. On the other hand, if our assumption that the probability of deriving w_X remains low without access to $\mathsf{San}(X)$ holds true, then the simulator will have a comparably low probability of finding a breach.

This choice of G and A may seem far-fetched, and utterly removed from realistic situations in which we are likely to care about privacy. What kind of "natural" auxiliary information would essentially serve as a ciphertext for exactly the secret we wish to learn? Consider the following example.

Example 20.4 *Let X be a database containing the number of working hours reported by each TA in the Spring 2017 semester, and let C return 1 when it receives a pair containing the name of a TA, and the number of hours they reported. Suppose that one of the instructors plays the role of G, by offhandedly commenting that "Sam worked 20% more than the average TA this semester!" Finally, let* San *return the average of the values in X.*

Notice that given $G(X)$ amd $\mathsf{San}(X)$, we can easily construct a winning breach. However, playing the role of a simulator who is only given $G(X)$, our probability of winning will remain low (assuming a somewhat uniform distribution of values in X). Although we didn't literarlly encrypt or decrypt anything, at least by any conventional notion of encryption, the background information contained in $G(X)$ effectively encodes the entire breach, but remains useless without the information returned by $\mathsf{San}(X)$.

Looking forward. Without covering all of the relevant technical details, this discussion demonstrates the impossibility of achieving a general notion of absolute privacy even when we impose mild conditions on what we expect in terms of utility, secrets, and background information. Dwork and Naor published a full version of this argument in 2006, and at the same time proposed a new general-purpose definition called *Differential Privacy* that has remained the de facto standard ever since. In the next lecture, we'll see what Differential Privacy means, and how it manages to remain useful despite the sort of impossibility result we covered today.