6.885: Advanced Topics in Data Processing Fall 2013

Lecture 20: Security and privacy for data processing

Stephen Tu

1 Overview

In this lecture, we will focus on how issues surrounding security and privacy come into play in data management systems and data mining algorithms. In an ideal world, we would not need to worry about these issues. But unfortunately, we do not live in such a world, so these are issues which are important to discuss.

1.1 Threat models

Before we even begin to discuss solutions, we need to first frame the problem. Once we frame the problem, we typically need to impose some assumptions (solutions which employ no assumptions are ofter either impractical or impossible). We will touch on two different problems, and then discuss a threat model for each. Roughly speaking, a *threat model* defines what an adversary is capable of doing (and not doing). We do not claim these specific problems to be all encompassing. Instead, we aim to provide a few instances of problems which actually occur in the real world.

Data processing over confidential data. The first problem we will consider is the issue of protecting the confidentiality of records stored in a database, while allowing for computation to run over the data. Note that the latter requirement is crucial, otherwise existing encryption schemes are more or less sufficient. Here, we assume that a particular organization has a collection of records (e.g. medical history, user click logs) and wants to store these records in a database. The organization is concerned that the database may be compromised by an adversary. Several instantiations of this include:

(a) An outside adversary hacks into the organization's computer which hosts the database, and downloads all the user data.

(b) A disgruntled employee is angry at his company and decides to download all company data onto his own USB drive.

(c) A small company, with not that much computing resources, decides to use a cloud service (e.g. Amazon EC2) to host its storage infrastructure. The cloud service is either malicious (e.g. sells hosted company data to other advertising agencies), or the cloud service is compromised.

For this problem, we will consider the following threat model illustrated in Figure 1: the organization has a local machine which is *trusted*. It runs a database on a remote machine which is *untrusted*. An adversary can observe all actions occurring on the untrusted machine (e.g. read all the bytes from memory and disk, sniff all the network traffic). It, however, cannot read the bytes

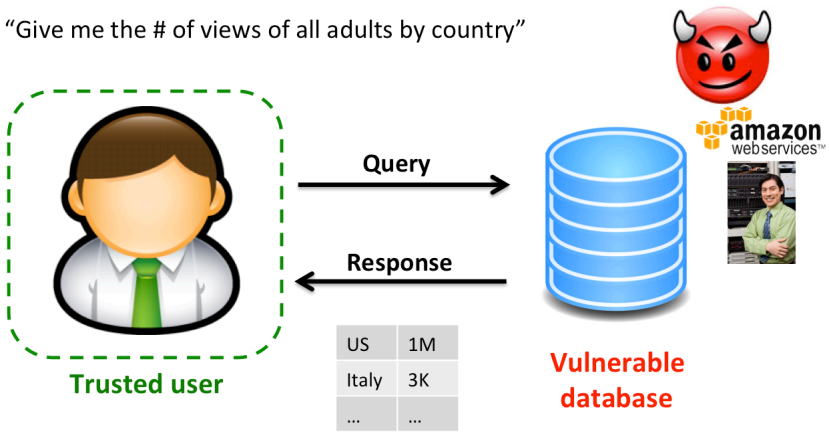

Figure 1: Threat model

on the trusted machine. For this scenario to be meaningful, we assume the remote machine to be much more powerful; the trusted machine needs to only store private cryptographic keys and run primitive cryptographic operations (such as decrypting a ciphertext using a private key).

Release of aggregate information. This problem is somewhat complementary to the previous one. In this problem, an organization has a database of records where the individual values are presumed to be sensitive. For example, the database could contain information like whether a patient has disease X, or what political affiliation a person has.

This organization, however, would like to release a summary, or aggregate, of the personal data. Using the previous example, this could be statistics such as how many people developed breast cancer in a certain year, or how many people are registered Republicans. These aggregate summaries of data are seemingly harmless; even if I know that there were 1000 new cases of cancer this year, I do not in particular learn that Bob in particular was diagnosed with cancer. However, one could easily imagine cases where aggregate information *does* reveal very personal information. A silly example is if a company has N employees (all paid the same salary) and a single boss (who is paid a higher salary), and one learns the average salary of the company, then one can easily infer the boss's salary from the average! One concrete instantiation of this scenario where aggregates over sensitive data are being released is the release of US census information.

The general problem is stated as: how does a trusted organization release the result of an aggregate computation over a private database publicly, while ensuring that each of the individual records in the database are not leaked as a result of the information release? The threat model in this case is simpler. An organization has a database which is trusted, and also runs computation which is trusted. An adversary can only observe auxiliary information in the world and can obviously learn the result of any computation run by the trusted party, but is not allowed to see the entire database contents.

2 Confidential query processing

In this section we focus on the first problem outlined, which is how to run a database on a server which is untrusted, while protecting the confidentiality of the data. Before proceeding, we briefly mention that we are not concerned with integrity here; we assume the adversary is passive (he can only observe all the data at the server) and not active (he cannot for instance change the code which runs on the server). While the passive assumption is not necessary for protecting data confidentiality, it is necessary to ensure utility (otherwise every query we ask the server cannot be assumed to have any meaning).

2.1 The first solution

Hacigumus et al. [5] was the first to bring this problem up to the database community. Their paper was quite visionary; in 2002, before "the cloud" was even a buzzword, they envisioned that in the future databases would be offered as a *service*. With this, they foresaw the need for security– how could a user of a database as a service platform ensure that confidentiality of data was not compromised by using the service?

Their solution assumes the same threat model we outlined previously, and works as follows. They place encrypted data on the untrusted server. They assume the existence of a symmetric key block cipher, such as AES. A query is executed by first rewriting it locally (on the trusted client), and issuing an rewritten query to the server. The (encrypted) results are sent back to the client, which decrypts to get the final answer.

Storage. More specifically, for each column in a table, Hacigumus partitions the domain of the column into k disjoint subsets which cover the domain. The trusted server maintains the mapping between each element in the domain and its corresponding partition (which Hacigumus denotes with a unique partition id). To keep the exposition simple, let us assume that the mappings preserve order (although this weakens security). Notationally, let table T have c columns $T_1, ..., T_c$, each with $P_1, ..., P_k$ partitions. Let $\text{dom}(T_i)$ denote the domain of the i-th column, $M_i : \text{dom}(T_i) \to [P_i]$ be the mapping from value to partition ID, and $\mathsf{Enc}_k(v)$ denote the encryption of v under key k. Hacigumus stores each record $r \in T$ as

$$\langle \mathsf{Enc}_{T.k}(r), M_1(r[1]), ..., M_c(r[c]) \rangle$$

where $T.k$ is the symmetric key for table T and $r[i]$ denotes the value of the i-th column of r. Notice that the partition identifies are stored in *plaintext*. This will enable Hacigumus to perform approximate filtering on the server, as we will see later. For concreteness, a mapping M_i might look like the one shown in Figure 2.

Query processing. With this in mind, query processing reduces to the task of translating predicates into partitions to select, issuing a query to the untrusted server to fetch the partitions to the (trusted) client, and performing the final filtering step on the client.

Let us consider the SQL query over a single table `employees`:

```
select *
```

Values	Partition
$[0, 50)$	0
$[50, 75)$	1
$[75, 80)$	2
$[80, 100)$	3

Figure 2: A possible mapping M_i for $\text{dom}(T_i) = [0, 100)$.

```
from employees
where
    salary > 50000 and
    dept in ('sales', 'hr');
```

Now suppose $M_{salary}(50000) = 10$, $M_{dept}(\text{sales}) = 15$, and $M_{dept}(\text{hr}) = 20$. Hacigumus then issues the following query to the untrusted server:

```
select *
from employees_enc
where
    salary_partid >= 10 and
    dept_partid in (15, 20);
```

It is not hard to see that any query which matches both predicates *must* be in this result set. The converse is obviously not true through, which is why the trusted client must do some additional post-processing to filter out *false* matches. However, since the client has the symmetric key k, it can simply decrypt all the values and filter locally.

From this exposition, it should be clear how to generalize this scheme to more complicated queries.

Security. There are several gaping security holes in this current exposition. First, the mappings M_i should be *securely* randomized (see discussion below), and distributed over a larger range of values (say over 2^{32}). To handle range queries, the mappings M_i must still preserve order (in order to support e.g. `col1 > col2`), but they do not need to be contiguous. If we know a certain column will never serve as a predicate, then we do not need to maintain a mapping for it (and do not need to leak which partition it belongs to when storing it).

Notice that in the limit (a single partition for each possible value in the domain), the problem of generating a non-order preserving mapping reduces to the problem of generating a pseudo-random permutation (PRP) [1] on $\text{dom}(T_i)$. Furthermore, the problem of generating an order-preserving mapping is simply the problem of order-preserving encryption (OPE) [3] on $\text{dom}(T_i)$. Even when we do not have a partition for each value, careful attention must be paid to properly generating the mappings. One of the major flaws with this work is that it does not provide guidelines for how to security generate partitions; without some background in cryptography, it is unclear how practitioners would actually implement such a system.

Drawbacks. There are also some systems level issues with this design. Note that, unless each column is partitioned exactly (one per value), then *every* query requires the client to download

more results than necessary and run post-processing filtering. For example, a primary key lookup, which we expect to be the fastest access method possible in a database, could potentially involve downloading and decrypting hundreds (if not more) of records.

Another big issue is with aggregates. For example, a

```
select sum(salary) from employees;
```

now requires transferring the *entire* employees table to the client, whereas in a plaintext database simply involves transferring a single number.

2.2 A more practical solution

CryptDB [7] is a recent system which extends this line of work and tries to address some of the shortcomings of [5]. CryptDB's main insight is to use specialized cryptosystems designed to perform common database operations entirely on the server. Before discussing CryptDB further, we make a small digression to talk about the properties of various cryptosystems. For those of you who are experts in cryptography, please excuse the oversimplification and lack of rigour in the following discussion.

Randomized encryption (RND). This is usually what people think of when they think of encryption. A randomized encryption scheme has the following property (usually known as *semantic security* or IND-CPA): if $a = \mathsf{EncRand}_k(x)$ and $b = \mathsf{EncRand}_k(x)$, then with high probability $a \neq b$. IND-CPA is achieved in practice by using a secure block cipher (e.g. AES-CBC) coupled with a randomly chosen initialization vector (IV).

Deterministic encryption (DET). This is a weaker form of encryption than randomized. Specifically, we have for all x, y in the plaintext space

$$x = y \Leftrightarrow \mathsf{EncDet}_k(x) = \mathsf{EncDet}_k(y)$$

A practical realization of this is to use AES-CBC as before, but set the IV to zero[1].

Order preserving encryption (OPE). This is an even weaker form of encryption than deterministic (OPE implies deterministic). Specifically, for all x, y we have

$$x < y \Leftrightarrow \mathsf{EncOPE}_k(x) < \mathsf{EncOPE}_k(y)$$

Boldyreva [3] is a provably secure cryptosystem which achieves this property.

Additive homomorphic encryption (HOM). An additive homomorphic encryption scheme has the following property: given any two ciphertexts $\mathsf{EncHOM}_k(x)$ and $\mathsf{EncHOM}_k(y)$, there exists some computable function f such that $f(\mathsf{EncHOM}_k(x), \mathsf{EncHOM}_k(y)) = \mathsf{EncHOM}_k(x + y)$. Note

[1] This has some security issues for values greater than 128 bits long. See [7] for more details

that f does not have access to k.[2] Paillier [6] is an asymmetric encryption scheme (that happens to be IND-CPA) that satisfies this property with f being multiplication modulo a public parameter.

With these cryptosystems in mind, we can discuss how CryptDB works. The high level idea is similar in spirit to [5]. CryptDB stores encrypted data on the server, and processes queries by rewriting queries to run over the encrypted database, fetching the encrypted results, and decrypting locally on the trusted client.

Storage. For each column, CryptDB uses (up to) four columns in the encrypted database to store an encryption of the value in multiple cryptosystems. This is shown in Figure 3.

Employees

ID	*Name*
23	Alice

Employees-enc

ID-IV	*ID-Eq*	*ID-Ord*	*ID-Add*	*Name-IV*	*Name-Eq*	*Name-Ord*	*Name-Search*
x27c3	x2b82	xcb94	xc2e4	x8a13	xd1e3	x7eb1	x29b0

Figure 3: CryptDB data layout on the untrusted server (original table shown on left).

Specifically, CryptDB uses an *onion* for each column, as shown in Figure 4. The idea is that each onion is dedicated towards a primitive operation. There are four onions because CryptDB has four main primitive operations: *equality testing*, *inequality testing*, *addition*, and *keyword searching*. Onions are a security feature. In principle, CryptDB could use the weakest (most functional, least

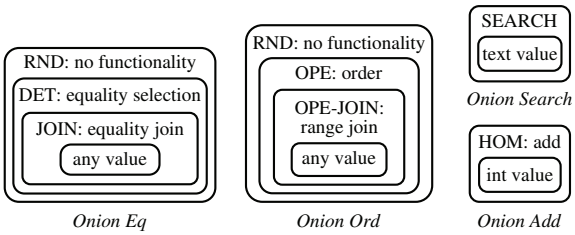

Figure 4: Onions in CryptDB. Each (logical) column is encrypted in up to four different onions.

secure) cryptosystem for each onion up front. But if a particular column is never used, for instance, in a `where` clause, it makes little sense security wise to reveal its order preserving encryption value.

Onions work by encrypting from less secure to more secure encryption schemes. To form the i-th layer of the onion, simply encrypt the $(i-1)$-th layer in the encryption scheme of the i-th layer (using a different key). The onion stops when it is encrypted by an IND-CPA scheme (random or Paillier). The neat trick with onions is that they can be *adjusted* dynamically. To adjust an onion, simply use the key of the i-th layer to decrypt, revealing the $(i-1)$-th layer.

Query processing. With the storage scheme in mind, query processing is quite simple. CryptDB simply rewrites every predicate to use the appropriate onion for the operation.

For example, if CryptDB sees the query

```
select * from employees where name = 'alice';
```

[2] In an asymmetric encryption scheme, f does not have access to the secret key.

it first checks to see if `name_eq` has been adjusted to reveal the deterministic layer. If not, it issues a user defined function (UDF) to the server to adjust the onion down to deterministic, giving the key used to encrypt the randomized layer. Once the onion is properly adjusted, CryptDB then issues the query

```
select * from employees_enc where name_eq = 0xab12df;
```

where `0xab12df` denotes $\mathsf{EncDet}_k(\text{alice})$.

More interestingly, if a user issues the query

```
select sum(salary) from employees;
```

then CryptDB uses a UDF called `paillier_sum` (which implements multiplication modulo n) to perform the sum homomorphically on the client. This looks like

```
select paillier_sum(salary_add, n) from employees_enc;
```

The result is a *single* Paillier ciphertext, not the entire database as in [5].

Equi-joins are easy to support in CryptDB if both columns share the same encryption key (which is the case if the workload is available beforehand). However, if this is not the case, CryptDB uses a cryptographic join adjustment based on elliptic curves to move both columns to the same key (which allows for equality comparisons). The details of this are beyond the scope of this lecture (see [7]).

Range scans, `ORDER BY`, `MIN`, and `MAX` are all easy to support efficiently with OPE. The neat thing about CryptDB's design is that it can leverage a lot of existing database machinery for performance. For instance, indices can be built on top of deterministic and order preserving columns (once the onions are unraveled). This means CryptDB can, for instance, build a secondary B-tree index on say a salaries column, and then scans for particular ranges of salaries become efficient.

Drawbacks. CryptDB's design is far from perfect. Onion adjustments are very expensive on big tables. This is mitigated by programmer assistance (e.g. use OPE on columns x,y,z) or workload trace analysis. However, once an onion is adjusted, it cannot be re-adjusted back to a stronger security scheme. Such a re-adjustment scheme would require performing all encryptions locally and updating all values at once (or in batches). It is also unclear when would be a right time to re-adjust onions.

Furthermore, CryptDB cannot support arbitrary SQL queries. For instance, any query involving multiplication by a non-constant (e.g. `select a*b from numbers`) cannot be supported because CryptDB does not use any cryptosystem which handles multiplications. The good news is as more specialized cryptosystems become available, they can be plugged into CryptDB fairly straightforwardly.

Actively being developed. CryptDB is being actively developed. The source code is here

```
git clone -b public git://g.csail.mit.edu/cryptdb
```

2.3 More related systems

There are a few other systems which aim to support query processing over encrypted data.

Monomi [8] extends CryptDB's basic approach, but focuses on handling queries which cannot be run by any of CryptDB's cryptosystems. The main idea here is to split the query to run both on the server and the client (much like [5]). Monomi presents a query optimizer to decide the best partitioning for a query given the physical database design. It also presents some work on automated database design for encrypted databases, building on a line of work for using integer linear programming for index selection.

Cipherbase [2] is a system which introduces encrypted query processing to Microsoft's SQL Server. The main difference between Cipherbase and CryptDB is that Cipherbase works with the assumption of trusted hardware on the untrusted server. This means there is a hardware unit which is capable of arbitrary computation on the server which is assumed to not be compromised. There is still overhead, however, of using the trusted hardware (data transfer). This is a really interesting design space; figuring out which computation to run where (client, server, hardware) is an interesting problem in optimizer design.

3 A side note: fully homomorphic encryption

Broadly speaking, fully homomorphic encryption (FHE) is the idea of being able to compute any function over encrypted data, without having access to the decryption keys. In other words, given an arbitrary f and $\mathsf{Enc}_k(x)$, FHE allows computation of $\mathsf{Enc}_k(f(x))$ without having access to k.

For more than thirty years, this was thought to be impossible, but Craig Gentry in 2009 published a fully homomorphic encryption scheme [4] which was a groundbreaking result in computer science.

Gentry's initial scheme was super impractical, but in the subsequent years many researches have improved the performance of FHE schemes by many orders of magnitude. It is to the point now where a single add or multiply can finish under a millisecond (this is very spectacular considering the initial overheads).

We are discussing FHE here, because in theory, FHE solves the problem that CryptDB and other systems are after– encrypted query processing! In other words, the function f can simply be the database query plan. Unfortunately, FHE is currently still too impractical to replace systems like CryptDB. This is because FHE only offers two primitives: *add* and *multiply*. In theory, every computable function can be decomposed into adds and multiplies (much like every function has an equivalent logic circuit). But in many types of non-trivial computation, circuit depth becomes quite unmanageable (imagine coding the database execution engine using only adds and multiplies!). For this reason, systems (such as CryptDB) which take advantage of specialized, but limited cryptosystems will still be relevant for the near future.

4 Secure release of aggregate information

The notes for this are in a separate note. See the course webpage for the link.

References

[1] How to construct pseudo-random permutations from pseudo-random functions. In *Advances in Cryptology*, volume 218 of *Lecture Notes in Computer Science*, pages 447–447. 1986.

[2] A. Arasu, S. Blanas, K. Eguro, M. Joglekar, R. Kaushik, D. Kossmann, R. Ramamurthy, P. Upadhyaya, and R. Venkatesan. Secure database-as-a-service with cipherbase. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1033–1036, 2013.

[3] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *Proceedings of the 28th Annual International Conference on Advances in Cryptology: The Theory and Applications of Cryptographic Techniques*, EUROCRYPT '09, 2009.

[4] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st Annual ACM Symposium on Theory of Computing*, STOC '09, pages 169–178, 2009.

[5] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing sql over encrypted data in the database-service-provider model. In *SIGMOD Conference*, pages 216–227, 2002.

[6] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques*, EUROCRYPT'99, pages 223–238, 1999.

[7] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. Cryptdb: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 85–100, 2011.

[8] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. In *Proceedings of the 39th international conference on Very Large Data Bases*, PVLDB'13, pages 289–300, 2013.