Implementation Assignment (Part 2)

Instructor: Tarik Moataz (tarik_moataz@brown.edu)

Overview

The purpose of this assignment is to implement two concrete instantiations for two advanced cryptographic primitives: (1) *private set intersection* (PSI) and (2) *structured encryption* (STE).

Private Set Intersection

PSI is a two-party secure computation protocol where each party holds a set of elements and neither wants to share its elements with the other party. At the end of the protocol, both parties will receive the intersection of the two sets without learning the remaining elements.[1]. PSI has received a lot of attention recently due to the many cases it can apply to. As an instance, PSI has been proposed as a tool to test human genomes, to perform contact list discovery or to measure conversion rates in online advertising. More concretely, consider the following example.

Use case. Consider a secure end-to-end encrypted messaging application (e.g., Signal) where the messages are encrypted end-to-end and the server cannot see the content of the messages. Imagine now that Alice wants to start using Signal so she first downloads the app, then she would like to know who from her contact list has Signal installed so she can talk to them. One naive approach is to send the entire contact list (i.e., phone numbers) to a central server which then performs an intersection to identify the corresponding contacts. Such a solution is quite intrusive as in order to participate in the system a server has to know your contact list.[2] A better approach is to use PSI between Alice and the server. On one hand, Alice has her contact list, and on the other hand the server has the list of all participants in the system.

Concrete instantiation. We consider a simple Diffie-Hellman based PSI [HFH99]. We consider two parties Alice $\mathcal{A}$ and Bob $\mathcal{B}$ who each owns a set S_A and S_B. The protocol works as follows:

1. $\mathcal{A}$ and $\mathcal{B}$ picks a random value α and β in $\mathbb{Z}_q$ where q is a prime number;
2. $\mathcal{A}$ parses her set $S_A = \{a_1, \cdots, a_m\}$ and sends to $\mathcal{B}$ the set
$$S'_A = \{H(a_1)^\alpha, \cdots, H(a_m)^\alpha\};$$
where H is a random oracle;
3. $\mathcal{B}$ parses his set $S_B = \{b_1, \cdots, b_n\}$ and sends to $\mathcal{A}$ the set
$$S'_B = \{H(b_1)^\beta, \cdots, H(b_n)^\beta\};$$

[1] Note that there are several variants of PSI where for example only one party receives the intersection.
[2] Note that this is not how Signal exactly works and this example is used here only for the sole purpose of illustration.

4. $\mathcal{A}$ receives then parses $S'_B = \{H(b_1)^\beta, \cdots, H(b_n)^\beta\}$ and computes

$$S''_B = \{(H(b_1)^\beta)^\alpha, \cdots, (H(b_n)^\beta)^\alpha\};$$

   $\mathcal{A}$ sends S''_B to $\mathcal{B}$ but also keeps a copy;

5. $\mathcal{B}$ receives then parses $S'_A = \{H(a_1)^\alpha, \cdots, H(a_m)^\alpha\}$ and computes

$$S''_A = \{(H(a_1)^\alpha)^\beta, \cdots, (H(a_m)^\alpha)^\beta\};$$

   $\mathcal{B}$ sends S''_A to $\mathcal{A}$ but also keeps a copy;

6. $\mathcal{A}$ (resp. $\mathcal{B}$) computes

$$\mathsf{Result} = S''_A \cap S''_B$$

One can easily verify that the final result is indeed the intersection of the two sets.[3] Note that both Alice and Bob need to keep track of the ordering in their respective set to be able to map back the elements in the result to the plaintext elements in S_A and S_B.

Task. Implement the PSI protocol above and run it on sizable sets (e.g., 1000 elements). The student can pick the programming language of her/his choice. Make sure that Alice and Bob are using different machines (or processes in a single machine). That is, the implementation should be designed in a client-server setting. The random oracle H can be instantiated using `SHA256`.

Structured Encryption.

Structured encryption (STE) is a cryptographic primitive that encrypts any arbitrary data structure in such a way that one can privately query it and update it. STE found several applications in data outsourcing such as the design of non-relation or relational end-to-end encrypted databases, encrypted file storage (also known as searchable symmetric encryption (SSE)), or encrypted graph databases. In the following, we will be interested in a specific type of STE called multi-map encryption schemes – an STE scheme for multi-map data structures (also called inverted indices).

Use case. In the following, we consider the design of an end-to-end encrypted file storage (e.g., end-to-end encrypted Google Drive)[4] where Alice has a set of documents that she wants to remotely store in the Cloud. If Alice was only interested to store the file securely in the cloud, then she can simply encrypt the files but this would negate any search functionality. Alice, instead, is looking for the functionality to store her documents encrypted but also to be able to search over them with the server knowing neither the content of her queries nor the content of her documents. Note that in the following, we will focus on the *single exact* keyword search setting in which Alice is only interested in fetching documents that contain a single keyword. There are other schemes in literature that can support more expressive queries such as Boolean queries or range queries.

[3] Note that the protocol above is a straw-man example and some details are omitted to simplify the description.

[4] Note that Google does not offer such a service and this example is solely intended for illustration purposes.

Concrete instantiation. We consider the construction by Cash et al. [CJJJKRS14]. We have two parties, Alice $\mathcal{A}$ and a server $\mathcal{S}$. Alice holds a set of documents $\mathbb{D} = (\mathbf{D_1}, \cdots, \mathbf{D_n})$. The protocol is divided in two phases: a setup phase and a query phase and we detail each below. The protocol makes use of a pseudo-random function $F : \{0,1\}^k \times \{0,1\}^m \to \{0,1\}^l$ and a symmetric encryption scheme $\mathsf{SKE} = (\mathsf{Enc}, \mathsf{Dec})$.

The **Setup** phase works as follows:

1. $\mathcal{A}$ instantiates an empty multi-map data structure MM;
2. $\mathcal{A}$ extracts all keywords from the document collection $\mathbb{D}$ and maps each keyword to all document identifiers that contain the keyword such that $\mathsf{MM}[w]$ will contain a tuple of document identifiers;
3. $\mathcal{A}$ generates two random keys $K_1, K_2 \xleftarrow{\$} \{0,1\}^k$;
4. $\mathcal{A}$ instantiates an empty dictionary structure DX and a list L;
5. for every keyword w,
   (a) compute $k_w = F(K_1, w)$;
   (b) initialize a counter cnt;
   (c) for every identifier id in $\mathsf{MM}[w]$, add the pair
   $$\Big(F(K_w, \mathsf{cnt}), \mathsf{SKE.Enc}(K_2, \mathsf{id})\Big)$$
   to L;
   (d) increment cnt;
6. randomly permute L and insert all pairs in the dictionary DX;
7. $\mathcal{A}$ sends DX to the server $\mathcal{S}$ and keeps the keys locally.

The **Query** phase works as follows:

1. $\mathcal{A}$ wants to query for a keyword w;
2. $\mathcal{A}$ computes $k_w = F(K_1, w)$ and sends it to $\mathcal{S}$;
3. $\mathcal{S}$ initializes a list L;
4. Until no result is found,
   (a) $\mathcal{S}$ computes $\ell = F(K_w, i)$;
   (b) if $\mathsf{DX}[\ell] \neq \perp$, $\mathcal{S}$ adds $\mathsf{DX}[\ell]$ to L;
   (c) $\mathcal{S}$ increments i;
5. $\mathcal{S}$ sends L to $\mathcal{A}$;
6. for each element ct in L, compute then output $\mathsf{SKE.Dec}(K_2, \mathrm{ct})$.

It is easy to see that the final output will be equal to $\mathsf{MM}[w]$. That is, returning all document identifiers that match a particular keyword.[5]

[5]Notice that in order to have a complete SSE scheme, there is a need to have an extra round to send the document identifiers and retrieve the encrypted documents. For simplicity we are not going to consider the entire protocol.

Task. Implement the multi-map encryption scheme above and run it on a sizable document collection (e.g., 1000 documents) – you can use a subset of the publicly available Enron dataset (`https://www.cs.cmu.edu/ enron/`). Similarly, the student can pick the programming language of her/his choice. Make sure that Alice and Bob are using different machines (or different processes in a single machine). That is, the implementation should be designed in a client-server setting. The pseudo-random function F can be instantiated using `HMAC-SHA256` and the encryption scheme can be instantiated using `AES` in counter mode. The length of the keys should be 32 bytes.

Deliverables

Each student needs to send an executable with a `README` detailing how to run the program. Make sure to add a link for the used dataset as it is going to be different from a student to another.
The deliverable has to be sent prior to **March, 26th** – one week after the end of the course.

Citations

1. B. Huberman, M. Franklin, and T. Hogg. *Enhancing privacy and trust in electronic communities*. In ACM Confer- ence on Electronic Commerce, pages 78–86, 1999.

2. D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. *Dynamic searchable encryption in very-large databases: Data structures and implementation*. In Network and Distributed System Security Symposium (NDSS '14), 2014.