6.892 Computing on Encrypted Data October 21, 2013

Lecture 6

Lecturer: Vinod Vaikuntanathan *Scribe: Aakanksha Sarda, Yilei Chen*

0.1 Recap

So far, we have covered ways to outsource computations privately, via Fully Homomorphic Encryption (FHE), and verifiably, via 1-round computationally secure probabilistically checkable proofs.

0.2 Overview

Today, we will motivate and define functional encryption (FE), and then describe a rudimentary "single-key" FE scheme based on Yao's Garbled Circuits.

0.3 Open problem

We previously described a general way of transforming a private-key FHE scheme into a public-key FHE scheme. The existence of such a transformation for FE is an open problem.

1 Motivation: Limitation of FHE

1.1 Motivating scenario for FHE

We motivated FHE by describing a "canonical" two-party scenario, where a "client" would like to privately outsource computation to an untrusted party (the "server"). In this scenario, the server doesn't need to do any action with the results of the computation. Now, two kinds of practical limitations arise with FHE: first, practical limitations of the motivating scenario (maybe we want the server to do some action based on the computation) and second, practical limitations of FHE even assuming the motivating scenario is applicable.

This is because, in the two-party FHE scheme the server never gains access to any cleartext: the input, final results and intermediate results are all encrypted.

1.2 Practical limitations of motivating scenario

One straightforward limitation of the motivating scenario is when we actually want the server to do an action based on the results of the computation. For example,

Encrypted Spam Filter: We would like the server to be able to run the spam-detection algorithm on encrypted emails and get the answer in clear-text. If the server can thus identify whether an encrypted message is spam or not, then the client doesn't have to download spam.

1.3 Practical limitations of FHE within motivating scenario

Note that even if we don't generally think of it as "results of a computation", any information derived from the input will be protected under FHE. So even in the our motivating scenario, where the server doesn't need access to any results *except to do the computation*, the efficiency of the computaiton could be affected, since the server can't follow data-dependant control flows. Some examples are:

- **Encrypted Turing Machines/C programs:** We would like to run a Turing Machine (or C program) on encrypted inputs in time proportional to the run-time on the same input in clear-text. This requires leaking at least the run-time of the TM on the clear-text input, which is not allowed under FHE!

- **Encrypted Binary Search:** This requires data-dependant loops, and following the control flow of the algorithm on unencrypted data would leak at least the position of the encrypted data in the array. Thus we can't do search in sublinear time with FHE, even on a sorted array.

2 Defining FE

Informally, Functional Encryption (FE) is an encryption scheme that can reveal the result (in the clear) of applying a function on an encrypted input without revealing any other information about the encrypted input.

Note that a scheme that reveals the result in the clear of applying any arbitrary function (for example, the identity function) to the input can't reasonably be thought of as secure. Similarly, if we had a scheme that revealed the run-time of any arbitrary Turing machines on an input, then a clever choice of function (for example, a function that runs in time m seconds for integer input m) would reveal everything about the encrypted input. So, instead, we will focus on schemes that reveal the result only for a specific function (or class of functions) or, analogously, schemes that support running only a specific Turing machine (or class of Turing machines) in input-specific run-time.

We follow [**?**] in to define function-specific FE.

2.1 Function-specific FE

Definition 1. *A Functional Encryption scheme for a class of functions $\mathcal{F}_n = \{f : \{0,1\}^n \to \{0,1\}\}$ is a tuple of Probabilistic Polynomial Time (PPT) algorithms (*`Setup, KeyGen, Enc, Dec`*) such that:*

- *The master key generation algorithm* `Setup` *takes as input the security parameter 1^λ and outputs a pair (*`pk, sk`*), i.e. public key and secret key.*
- *The function-specific key generation algorithm* `KeyGen` *takes as input the secret key* `sk` *and a function $f \in \mathcal{F}_n$ and ouputs a function-specific secret key $\mathtt{sk}_f$.*
- *The encryption algorithm* `Enc` *takes as input the public key* `pk` *and the message $m \in \{0,1\}^n$ and outputs a ciphertext c.*
- *The input-specific decryption algorithm* `Dec` *takes as input a ciphertext c (corresponding to a message m) and a function-specific secret key $\mathtt{sk}_f$ (corresponding to a function f) and outputs (in the clear) the result $f(m)$ of applying the function on the message.*

We require that:
(Correctness.) *For every message $m \in \{0,1\}^n$ and function $f \in \mathcal{F}_n$ we have:*

$$\Pr\left[r = f(m) \;\middle|\; \begin{array}{ccc} (\mathtt{pk},\mathtt{sk}) & \leftarrow & \mathtt{Setup}(1^\lambda) \\ \mathtt{sk}_f & \leftarrow & \mathtt{KeyGen}(\mathtt{sk},f) \\ c & \leftarrow & \mathtt{Enc}(\mathtt{pk},m) \\ r & \leftarrow & \mathtt{Dec}(c,\mathtt{sk}_f) \end{array} \right] = 1 - \mathtt{neg}(\lambda)$$

Where the probability is taken over the randomness used by `Setup, KeyGen, Enc` *and* `Dec`*.*

(Security.) *We require that for every (oracle) PPT adversary $\mathcal{A}$ and (oracle) PPT algorithm $\mathcal{M}$, there exists an (oracle) PPT $\mathcal{S}$ such that the following two distributions are indistinguishable:*

Real distribution:

1. `(pk, sk)` $\leftarrow$ `Setup`(1^λ)
2. $\mathtt{sk}_f \leftarrow$ `KeyGen`$(\mathtt{sk},f)$ *for several $f \in \mathcal{F}_n$ chosen by $\mathcal{A}$*
3. $(m_1, m_2, \ldots m_k) \leftarrow \mathcal{M}$ *such that $m_i \in \{0,1\}^n, \forall i \in [1, 2 \ldots k]$.*

4. $c_i \leftarrow \texttt{Enc}(\texttt{pk}, m_i), \forall i \in [1, 2 \dots k]$.

5. $\texttt{sk}_f \leftarrow \texttt{KeyGen}(\texttt{sk}, f)$ *for several* $f \in \mathcal{F}_n$ *adaptively chosen by* $\mathcal{A}$

6. $\alpha \leftarrow \mathcal{A}(\texttt{pk}, \texttt{sk}_{f1} \dots \texttt{sk}_{fj}, c_1 \dots c_k)$

7. *Output*$(f_1, f_2 \dots f_j, x_1, x_2 \dots x_k, \alpha)$

Ideal distribution:

1. $(m_1, m_2, \dots m_k) \leftarrow \mathcal{M}$ *such that* $m_i \in \{0,1\}^n, \forall i \in [1, 2 \dots k]$.

2. $f_j(m_1 \dots m_k)$ *for several* $f_j \in \mathcal{F}_n$ *adaptively chosen by* $\mathcal{A}$

3. $\alpha \leftarrow \mathcal{S}(n, f_1(x_1) \dots f_j(x_1) \dots f_j(x_k))$

4. *Output* $(f_1, f_2 \dots f_j, x_1, x_2 \dots x_k, \alpha)$

Informally, the security requirement means that we require the adversary to not gain any additional information over what we explicitly want to give it.

So, we consider the capabilities of the adversary, in the real world, who follows the protocol. Thus, in addition to the answers $f_1(m_1) \dots f_j(m_1) \dots f_j(m_k)$ that we explicitly want to give it, the adversary also gets access to various ciphertexts and function-specific secret keys. We now compare the capabilities of this adversary in the real world to the capabilities of a simulator in an ideal world.

In the ideal world, we only give the simulator access to the answers $f_1(m_1) \dots f_j(m_1) \dots f_j(m_k)$, that we explicitly wanted to give, and nothing else. Note that we don't use any encryption in this ideal world - it just corresponds to the best case scenario where the simulator gets nothing (not even seemingly irrelevant things) besides what we wanted to explicitly give it.

Now, if we can prove that the (computationally bounded) adversary in the real world gains no additional capabilities (as manifested in its output α) over the simulator that gets the same - i.e. the two Ouput distributions are computationally indistinguishable - then we say that the functional encryption scheme is secure.

2.2 A solution for the Turing Machine example

Consider the desiderata of Functional Encryption:

1. Collusion resistance: For any two party with different secret keys on their own, even if they collude, they still cannot learn more. (Many key security)

2. Succinctness: Running time of Encryption and the length of ciphertext $|c|$ equals poly$(|x|, \lambda)$, and it doesn't depend on the complexity of the computation (i.e. the circuit size).

Let's start with a construction with the functional encryption defined above, for the Turing Machine example. We might revise the revised goal for a moment: we allow the server to have a heavy preprocessing phase - the server can do whatever [1] it wants before getting the input. But once the input comes, the server should run in time proportional to the input size, and to the actual running time. The preprocessing phase take the Turing Machine as a input. The running time is proportional to the running time of Turing Machine.

If we have a functional encryption scheme, plus FHE, which is a natural starting point since the semantic security of FHE provides the desired security that nothing leaks about input x. We encrypt the input x using FHE, write the Turing Machine as a circuit[2]. But the size of the circuit is proportional to the worst case running time, which is exactly what we want to get around.

[1] Do as long as the worst case running time is in polynomial.
[2] Since FHE works on circuit.

As we said in the beginning, to achieve this goal we have to leak some information of x. In some applications, leaking the running time of some input is not that bad. Let's just leak the running time t of machine M on x ($t = M(x)$), but nothing more. The idea is to use functional encryption to enable the server to determine at various intermediary steps in the evaluation whether the computation is finished. For each intermediary step, the client provides a secret key for a function that returns a bit indicating whether the computation finishes or not.

However, if the client provides a key for each step, then the amount of keys is still proportional to the worst-case running time T_{worst}. But now the client could get rid of that by choosing intermediate steps which increase exponentially, say, providing the secret keys $sk_{M_1}, sk_{M_2}, sk_{M_4}, ...$ for the machines [3] that halt in $1, 2, 4, ..., 2^i, ..., 2^{\lceil \log T_{worst} \rceil}$ steps, each key being generated with a different master secret key. The work in the preprocessing phase is at least $O(T_{worst})$, which is costly but once and for all.

In the online phase, the client provides FE encryptions of (M, x) and an FHE ciphertext $(\tilde{M}, \tilde{x})$ for (M, x), to be used for a separate FHE evaluation. The server learns the smallest t for which the computation of M on x stops in 2^t steps, by the keys generated in the preprocessing phase. Then homomorphically evaluate $(\tilde{M}, \tilde{x})$ by $C_t(\cdot)$, which is the (universal) circuit that runs M on t steps. Notice that $|C_t(x) = \text{poly}(t)|$, i.e. the circuit runs in time polynomial in the runtime of M on x.

3 A construction of FE with single-key security by Yao's Garbled Circuits

We will start with a simple and rudimentary construction of functional encryption, which will give us one-key security.

Before we start, let's think how to construct FE for a function family with polynomially many functions. It turns out there's a trivial construction that you can just encrypt the result of all the f on input x. However, if the function family has exponentially many functions, then the problem of the above construction is, when you encrypt the input, you don't know which function is it going to be evaluated. Hence a function family with exponentially many function is a non-trivial task.

To achieve one-key security of such a function family, where one-key security means the scheme is secure as long as the adversary ask for the secret key for one function, we will use Yao's Garbled Circuits.

There are different ways to describe the garbled circuits, but we will focus on a construction that is motivated by functional encryption - we have a circuit C (of some function f), and want to do outsource computation. We will describe the construction in the offline-online manner:

1. Offline phase: client got circuit C, garble it as $\mathsf{GC}, sk = \mathsf{Garbled.Ckt}(1^\lambda, C)$, send GC to the server.
2. Online phase: client garble the input x as $\mathsf{gx} = \mathsf{Garbled.Input}(1^\lambda, x, sk)$ send gx to the server.
3. Server evaluate as $y = \mathsf{Garbled.Eval}(\mathsf{GC}, \mathsf{gx})$.

In the garbled scheme presented by Yao, the secret key is of the form $sk = \{L_i^0, L_i^1\}_{i=1}^n$, and the encoding of an input x of n bits is of the form $(L_1^{x_1}, ..., L_n^{x_n})$, where x_i is the i-th bit of x.

We need the correctness guarantee that y should be equal to $C(x)$ with high probability. Also we require efficiency that the evaluation runs in time specific to the input.

Two security guarantees are of concern: the input privacy and circuit privacy. For Yao's garbled scheme, the security holds only for one-time evaluation of the circuit, since if the adversary receive more than one encoding of input, it breaks these security guarantees.

Formally, the security is defined as: there exists a PPT simulator Sim: $(\tilde{\mathsf{GC}}, \tilde{\mathsf{gx}}) = \mathsf{Sim}(C(x), 1^\lambda, 1^{|C|}, 1^{|x|})$, that is computationally indistinguishable from the real garbled circuit and the garbled input:

$$(\mathsf{GC}, \mathsf{gx}) \approx_c (\tilde{\mathsf{GC}}, \tilde{\mathsf{gx}})$$

[3] Here we would think the machines as Universal Turing Machines, which takes M and an input x, runs M on x for 2^i steps. Therefore it's not limited with an a priori fixed number of Turing Machines. The simulation could be done in polynomial time.

Here we show an example of how to construct the scheme above for the NAND gate [4].

$$\mathsf{GC}_{\mathsf{NAND}} = \begin{cases} Enc_{L_0^0}(Enc_{L_1^0}(L_{out}^1)) \\ Enc_{L_0^0}(Enc_{L_1^1}(L_{out}^1)) \\ Enc_{L_0^1}(Enc_{L_1^0}(L_{out}^1)) \\ Enc_{L_0^1}(Enc_{L_1^1}(L_{out}^0)) \end{cases}$$

The garbled circuit is represent by a table, which is generated by the client in the offline phase, and send to the server. Each pair of input garblings serves as a key to the encryption of the corresponding output garbling. In the online phase, for example, if the client receive the input $x = (0, 1)$, he gives the server $\mathsf{gx} = (L_0^0, L_1^1)$. The server evaluate as the following: he attempts to decrypt each row of the table, but only one decryption will succeed, which will reveal the output wire secret L_{out}, which is L_{out}^1 in our example.

If there are more gates (all boolean functions could be a combination of NAND gates), the server then proceeds evaluating the entire garbled circuit as above, and eventually obtains the result.

We will formally prove security in the next lecture.

References

[BSW11] Boneh, Dan, Amit Sahai, and Brent Waters. "Functional encryption: Definitions and challenges." Theory of Cryptography. Springer Berlin Heidelberg, 2011. 253-273.

[GKPVZ12] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan and Nickolai Zeldovich. "Reusable Garbled Circuits and Succinct Functional Encryption." Cryptology ePrint Archive, Report 2012/733, 2012.

[4] In addition, we need to pick a permutation of the table to hide the input.