Introduction to Cryptography April 24, 2018

Lecture 22: Secure Computation I

Instructor: Vipul Goyal *Scribe: David Gershuni*

1 Introduction

Secure Computation AKA Secure Multi-party Computation is used when multiple parties do not trust each other, but they wish to compute some function of their collective inputs, and critically, they do not want to reveal any other information to each other other than the output of the function on their combined inputs.

We'll make this definition more formal in a moment, but first, let's consider an example that will make this more concrete. This example is one of the original Secure Multi-party Computation problems, and it is known as Yao's Millionaire Problem.

1.1 Example: Yao's Millionaire Problem

We have two millionaires A and B, and they wish to know which of them has more money. But neither millionaire is willing to reveal how much money he or she has. Secure Multi-party Computation will enable A and B to exchange messages such that they both will know who has the higher net worth, but neither will learn the other's net worth.

2 The Setting for Secure Multi-party Computation

We have n parties who do not trust each other, $(P_1, P_2, ..., P_n)$, each with a private input, $(x_1, x_2, ..., x_n)$ and a single trusted, public function f shared by all. These n parties wish to jointly compute $f(x_1, x_2, ..., x_n)$ without revealing anything to each other other than the output of f.

Now consider Yao's Millionaire Problem again. Here, $n = 2$, so our two parties P_1 and P_2 are the millionaires, and the inputs (x_1, x_2) are their respective net worths. The computed output is $f(x_1, x_2)$ where f is a function defined by $f(x_1, x_2) = \begin{cases} P_1 & x_1 > x_2 \\ P_2 & x_1 < x_2 \\ Equal & otherwise \end{cases}$

2.1 Real-world Applications

Situations that require Secure Multi-party Computation arise often in real-life, for example:

- **National security:** Airlines wish to cross-reference each passenger's name and passport number with the Department of Homeland Security's terrorist watch list, but the DHS does not wish to reveal its watch-list and the airlines do not wish to expose innocent passengers' travel plans to the DHS.
- **Privacy-preserving data mining:** A groups of several independnt hospitals wish to collaboratively compute statistics about the most effective treatments from their patient data

and the data of fellow hospitals, but they cannot directly exchange patient records due to privacy laws. Using SMP, they can compute these functions without directly revealing anything patient records.

Note: Secure Multi-party Computation is also known as Multi-party Computation or MPC. When $n = 2$, it's called Secure 2-Party Computation or 2PC.

2.2 Example: Zero Knowledge Proofs as an instance of MPC

Almost any cryptographic problem can be framed as an instance of MPC. For example, Zero Knowledge Proofs are an instance of MPC. Consider the case of Zero Knowledge 3-coloring on graphs. In this case, $n = 2$. Our two parties P_1 and P_2 are the prover and the verifier, respectively. The input x_1 is the witness of the 3-coloring, and x_2 is null (denoted $\perp$). The computed output is $f(x_1, x_2)$ where f is a function that returns 1 if x_1 is a valid witness and 0 otherwise.

Note: for the rest of this lecture, we'll restrict ourselves to Secure 2-party Computation only. This will make the discussion simpler to understand.

3 Defining Security in Multi-party Computation

We'll construct a definition of security for MPC that is similar to the one for ZKP. In ZKP, the simulator produces a simulated view (i.e. a transcript) that is indistinguishable from a real view *given nothing as input*. Since a simulator is able to do this given nothing, we conclude that the protocol doesn't leak anything at all.

The natural extension to MPC is that the MPC simulator has to produce a view that is indistinguishable from a real MPC view given *only the function output*. Then we conclude that the MPC protocol doesn't leak anything (except the function output).

Definition 1 *Security in MPC*

Let P_1, P_2 be our 2 parties as before, and let P_b be the corrupt party. Let $View_{real}^{P_b}$ be the distribution of the views of P_b in the real execution of the protocol. And let S^{P_b} be a PPT simulator of the view of P_b. An MPC protocol is secure if

$$\forall (x_1, x_2),\ \exists\, S^{P_b}\ s.t.\ \{\, View_{real}^{P_b} \,\} \approx_c \{\, S^{P_b} \,\}$$

where the simulator S^{P_b} is given access to the protocol and to an oracle that takes an input x_1 and returns $f(x_1, x_2)$. S^{P_b} is allowed to query the oracle exactly once per execution of the protocol. The oracle is said to operate with 'ideal functionality.'

Remark 1 *Note that this definition captures the requirement that the dishonest party can only learn $f(x_1, x_2)$ (privacy) but it fails to capture the requirement that the honest parties should get correct output (correctness). We'll stick to this simpler definition for this lecture.*

4 The 2 Types of Adversaries in MPC

There are 2 types of adversaries we'll try to secure our protocol against, 'honest but curious' adversaries and 'malicious' adversaries.'

4.1 Honest but Curious Adversaries (AKA Semi-honest Adversaries)

An 'honest but curious adversary' follows all instructions of the protocol faithfully during its execution. That is, it does not cheat at any step of the protocol's execution. However, it may later attempt to perform dishonest tasks, using the transcript of the protocol after the fact.

This weaker adversary model will be used as a stepping stone to build MPC against fully dishonest parties later. It can also be used to represent a party that was corrupted or compromised after the protocol was executed honestly.

4.2 Malicious Adversaries (AKA Fully Dishonest Adversaries)

A malicious adversary operates without any limits on its dishonesty. We can make no assumptions about whether or not it will faithfully execute any steps of the protocol. Its only restriction is that is must be PPT.

5 Oblivious Transfer

We'll now describe a Secure 2-party Computation protocol called Oblivious Transfer that is secure only against semi-dishonest adversaries. It allows a party to request and receive exactly 1 of 2 bits from another party without the other party knowing which bit it sent.

5.1 Overview of Oblivious Transfer

1. *We have two parties, A and B.*
2. *A has a private pair of bits (a_0, a_1).*
3. *B has a single bit b.*
4. *A and B exchanges messages such that B learns only a_b ($a_{\bar{b}}$ remains secret) while A learns nothing.*
5. *B's final output is a_b.*
6. *A's final output is $\perp$ (nothing).*

In order to construct an oblivious transfer protocol, we'll use a trapdoor one-way permutation. Recall the definition of a trapdoor one-way permutation:

Definition 2 *Trapdoor Permutation (Trapdoor OWP) A collection of trapdoor permutations is a family of permutations $\mathscr{F} = \{f_i : D_i \to R_i\}$ for $i \in I$ that satisfies the following properties:*

- ***Sampling function:*** *$\exists$ a PPT Gen s.t. $Gen(1^n)$ outputs $(i, t) \in I$*
- ***Sampling from domain:*** *$\exists$ a PPT algorithm that on input i outputs a uniformly random element of D_i*
- ***Evaluation:*** *$\exists$ a PPT algorithm that on input $i, x \in D_i$ outputs $f_i(x)$*

- ***Hard to invert:*** *$\forall$ non-uniform PPT adversary A, $\exists$ a negligible function $\mu(.)$ s.t. :*

$$Pr[i \leftarrow Gen(1^n), x \leftarrow D_i, y \leftarrow f_i(x) : f_i(A(1^n, i, y)) = y] \leq \mu(n)$$

- ***Inversion with trapdoor:*** *$\exists$ a PPT algorithm that given (i, t, y) outputs $f_i^{-1}(y)$. More formally,*

$$\forall y \in R_i,\ Pr[\, f_i^{-1}(t, y) = x : f_i(x) = y\,] = 1$$

That is, if we know t, we can invert every image in the range.

5.2 Oblivious Transfer Protocol Using Trapdoor Permutations

Now we'll describe how to use trapdoor permutations to implement oblivious transfer.

1. *A samples (f, f^{-1}) from $\mathscr{F}$, a family of trapdoor permutations.*
   A sends only f to B.
2. *B samples x from the domain of f; B samples y from the range f; B sets $y_b = f(x)$ and $y_{\bar{b}} = y$. B sends (y_0, y_1) to A.*
3. *Using the trapdoor function f^{-1} and a hardcore predicate h of f, A computes:*
   $z_0 = h(f^{-1}(y_0)) \oplus a_0$ and $z_1 = h(f^{-1}(y_1)) \oplus a_1$.
   A sends (z_0, z_1) to B.
4. *B outputs $z_b \oplus h(x) = a_b$*
5. *A outputs $\perp$*

5.2.1 Example Execution of the Protocol

Say B picks $b = 0$. Then $y_0 = f(x)$. So $x = f^{-1}(y_0)$. Thus, when A returns (z_0, z_1), B will compute $z_0 \oplus h(x) = (h(f^{-1}(y_0)) \oplus a_0) \oplus h(x) = (h(x) \oplus a_0) \oplus h(x) = a_0$.

5.3 Proof of Security

Proof. *There are two cases: one in which A is corrupt (semi-honest) and one in which B is corrupt (semi-honest). Since this protocol is asymmetric, we'll have to construct a separate simulator for each case.*

5.3.1 Case 1: A is corrupt and B is honest.

For this case, we construct a simulator S^A. Our intent is to show that A's view is essentially zero knowledge.

We construct the simulator as follows:

1. *Sample (f, f^{-1}) honestly from $\mathscr{F}$. Then output f as the first message of the transcript.*
2. *Sample (y_0, y_1) randomly from the range of f.*
3. *Compute (z_0, z_1) honestly from f^{-1} as described in the protocol and output (z_0, z_1).*

Lemma 1 *The distribution of $\{S^A\}$ is identical to the distributions of A's view when the protocol is executed in the real world.*

Proof. *The views generated by this simulator are completely identical to the views generated by A in the actual protocol except for how (y_0, y_1) are chosen. Namely, B chooses x from the domain of f instead of y from the range of f. However, this difference does not change the distribution of (y_0, y_1) because f is a permutation. Therefore, these views are identical.*

5.3.2 Case 2: B is corrupt and A is honest.

In this case, we construct the simulator S^B, which only has input b. B does not have (a_0, a_1). Instead it queries an oracle, giving b as input and receives a_b back. We construct S^B so that B does not learn $a_{\bar{b}}$.

The simulator is constructed as follows:

1. *Sample (f, f^{-1}) from $\mathscr{F}$ honestly and output f.*
2. *Construct (y_0, y_1) honestly, using b as described in the protocol.*
3. *The simulator cannot compute the message (z_0, z_1) honestly because it doesn't know $a_{\bar{b}}$, so it samples $a'_{\bar{b}}$ randomly from $\{0, 1\}$ and uses it instead. Then it computes $z_b = h(f^{-1}(y_b)) \oplus a_b$ and $z_{\bar{b}} = h(f^{-1}(y_{\bar{b}})) \oplus a'_{\bar{b}}$ and outputs (z_0, z_1).*

Notice that neither transcript contains f^{-1}.

Lemma 2 *The distribution of $\{S^B\}$ is computationally indistinguishable from the distributions of B's view when the protocol is executed in the real world.*

Proof. *In order to distinguish the distribution of S^B from the distribution of real views of B, one must identify that $z_{\bar{b}}$ is incorrect. But because $z_{\bar{b}}$ is generated by XOR'ing $a'_{\bar{b}}$ with the hardcore bit $h(f^{-1}(y_{\bar{b}}))$, no PPT adversary can predict $z_{\bar{b}}$'s true value with greater than 1/2 probability. Doing so would contradict the definition of a hardcore bit. Therefore, the distribution of views of S^B is indistinguishable from the distribution of real views. With this lemma proven, the proof is complete.* ■