**CS 294 – Secure Computation** January 19, 2016

# Lectures 1&2: Introduction to Secure Computation, Yao's and GMW Protocols

*Instructor: Sanjam Garg* *Scribe: Pratyush Mishra*

## 1 Introduction

Secure multiparty computation considers the problem of different parties computing a joint function of their separate, private inputs without revealing any extra information about these inputs than that is leaked by just the result of the computation. This setting is well motivated, and captures many different applications. Considering some of these applications will provide intuition about how security should be defined for secure computation:

**Voting:** Electronic voting can be thought of as a multi party computation between $n$ players: the voters. Their input is their choice $b \in \{0, 1\}$ (we restrict ourselves to the binary choice setting wlog), and the function they wish to compute is the majority function.

Now consider what happens when only one user votes: their input is trivially revealed as the output of the computation. What does privacy of inputs mean in this scenario?

**Searchable Encryption:** Searchable encryption schemes allow clients to store their data with a server, and subsequently grant servers tokens to conduct specific searches. However, most schemes do not consider access pattern leakage. This leakage tells the server potentially valuable information about the underlying plaintext. How do we model all the different kinds information that is leaked?

From these examples we see that defining security is tricky, with lots of potential edge cases to consider. We want to ensure that no party can learn anything more from the secure computation protocol than it can from just its input and the result of the computation. To formalize this, we adopt the **real/ideal paradigm**.

### 1.1 Real/Ideal Paradigm

**Notation.** For simplicity, we stick to just two parties, $P_1$ and $P_2$. The input of each party is denoted by $x_1$ and $x_2$ respectively. These parties wish to compute a joint function $f(\cdot, \cdot)$ of their inputs.

**Real World.** In the real world, the two parties execute a protocol $\Pi$ to compute the function $f$. This protocol can involve multiple rounds of interaction. Each party can additionally have some randomness $r_1$ and $r_2$. The real world adversary $\mathcal{A}$ can corrupt at most one of these parties.

**Ideal World.** In the ideal world, a trusted party helps in the computation of $f$: each party sends their input to the trusted party and receives the output of the computation ($f(x_1, x_2)$). Here, the ideal world adversary $\mathcal{S}$ can again corrupt at most one party. Denote this entire protocol by $F$.

**Definition of Security.** We require that the views of the parties in each of the scenarios be identical, i.e. that a real-world execution of the protocol $\Pi$ should not leak any information not leaked by the ideal-world execution. Hence, the parties can only learn what they can infer from their inputs and the output $f(x_1, x_2)$. More formally, assuming $\mathcal{A}$ corrupts one party (say $P_1$, wlog), we define random variables $\mathsf{Real}_{\Pi,\mathcal{A}}(x_1, x_2) = \{x_1, r_1, \text{messages sent in } \Pi\}$ and $\mathsf{Ideal}_{F,\mathcal{S}}(x_1, x_2) = \{x_1, f(x_1, x_2)\}$. These random variables represent the views of the adversary in each of the two settings. Our definition of security thus requires that

$$\mathsf{Real}_{\Pi,\mathcal{A}}(x_1, x_2) \stackrel{c}{=} \mathsf{Ideal}_{F,\mathcal{S}}(x_1, x_2)$$

**Assumptions.** We have brushed over some details of the above setting. Below we state these assumptions explicitly:

1. **Communication channel:** We assume that the communication channel between the involved parties is completely insecure i.e. it does not preserve the privacy of the messages. However, we assume that it is reliable, which means that the adversary can drop messages, but if a message is delivered, then the receiver knows the origin.

2. **Corruption model:** Ww have different models of how and when the adversary can corrupt parties involved in the protocol:

   - *Static:* The adversary chooses which parties to corrupt before the protocol execution starts, and during the protocol, the malicious parties remain fixed.
   - *Adaptive:* The adversary can corrupt parties dynamically during the protocol execution, but the simulator can do the same.
   - *Mobile:* Parties corrupted by the adversary can be "uncorrupted" at any time during the protocol execution at the adversary's discretion.

3. **Fairness:** The protocols we consider are not "fair", i.e. the adversary can cause corrupted parties to abort arbitrarily. This can mean that one party does not get its share of the output of the computation.

4. **Bounds on corruption:** In some scenarios, we place upper bounds on the number of parties that the adversary can corrupt.

5. **Power of the adversary:** We consider primarily two types of adversaries:

   - *Semi-honest adversaries:* Corrupted parties follow the protocol execution honestly, but attempt to learn as much information as they can from the protocol transcript.
   - *Malicious adversaries:* Corrupted parties can deviate arbitrarily from the protocol.

6. **Standalone vs. Multiple execution:** In some settings, protocols can be executed in isolation; only one instance of a particular protocol is ever executed at any given time. In other settings, many different protocols can be executed concurrently. This can compromise security.

## 2 Yao's Two Party Computation Protocol

Yao's Two Party Protocol is a protocol conducted between two parties for computing any function. It is obtained by combining two primitives: a scheme for garbling circuits and oblivious transfer. Informally, the idea is that one party (say $P_1$) *garbles* a circuit. This involves assigning random labels to each wire of the circuit, including the input and output wires. $P_1$ then sends the garbled circuit and the labels corresponding to its input wires to $P_2$. The two parties then engage in an oblivious transfer protocol to transfer the labels corresponding to $P_2$'s inputs to $P_2$. $P_2$ then evaluates the garbled circuit using the two sets of input labels to get the result of the computation.

**Problem Setup:** Let $G$ and $W$ be the gates and wires respectively in $\mathcal{U}$, the universal circuit. Let $w_i$ be the $i^{th}$ input wire, $i \in [n]$, and $w_{\mathsf{out}}$ be the output wire.

**Definition 1 (Garbling Scheme)** *A* garbling scheme *is a pair of ppt. algorithms* $(\mathsf{Garble}, \mathsf{Eval})$*:*

- $\mathsf{Garble}(1^\kappa, C) \to \left(\widetilde{C}, \{\mathsf{lab}_{i,b_i}\}_{i\in[n], b_i\in\{0,1\}}\right)$. *The circuit $C$ has $n$ input wires and one output wire.*
- $\mathsf{Eval}(1^\kappa, \widetilde{C}, \{\mathsf{lab}_{i,x_i}\}_{i\in[n]}) \to y$.

*It satisfies the following two properties:*

**Correctness:** $\forall\ C, x$, *we have that*

$$\Pr[(\widetilde{C}, \mathsf{lab}_{i,b_i}) \leftarrow \mathsf{Garble}(1^\kappa, C) \wedge y \leftarrow \mathsf{Eval}(1^\kappa, \widetilde{C}, \mathsf{lab}_{i,x_i}) \wedge y \neq C(x)] = 0$$

**Security:** $\exists$ *simulator* $\mathcal{S}$ *s.t.* $\forall\ C, x$, *we have that*

$$(\widetilde{C}, \{\mathsf{lab}_{i,x_i}\}) \stackrel{c}{=} \mathcal{S}(1^\kappa, C(x))$$

### 2.1 Construction:

We construct a garbling scheme:

$\mathsf{Garble}(1^\kappa, C)$:
1. For each $w \in W$, sample $(k_w^0, k_w^1)$ as enc. keys.
2. For each $g \in G$ with input wires $w_0, w_1$, and output wire $w_2$, set

$$e_g := \left(\mathsf{Enc}_{k_{w_0}^a}\left(\mathsf{Enc}_{k_{w_1}^b}\left(0^\kappa || k_{w_2}^{g(a,b)}\right)\right)\right)_{a,b\in\{0,1\}}$$

3. Set $\widetilde{C} := \left((e_g)_{g\in G}, k_{w_{\mathsf{out}}}^0 \to 0, k_{w_{\mathsf{out}}}^1 \to 1\right)$.
4. Set $\mathsf{lab}_{i,b_i} := k_{w_i}^{b_i}$.
5. Output $(\widetilde{C}, \mathsf{lab}_{i,b_i})$.

$\mathsf{Eval}(1^\kappa, \widetilde{C}, \{\mathsf{lab}_{i,x_i}\}_{i\in[n]})$:
1. For each gate $g \in G$, obtain $\mathsf{lab}_{w_2} \leftarrow \mathsf{Dec}_{\mathsf{lab}_{w_0}}(\mathsf{Dec}_{\mathsf{lab}_{w_1}}(e_g))$.

**Figure 1:** Definition of a garbling scheme

### 2.2 Proof of Security

**Proof.** We construct a ppt. simulator $\mathsf{Sim}$ such that $\forall x \in \{0,1\}^n$:

$$\left\{\widetilde{C}, \mathsf{lab}_{i,x_i}\right\} \stackrel{c}{=} \{\mathsf{Sim}(\kappa, C(x))\}$$

$\mathsf{Sim}(1^\kappa, C(x))$:
1. Sample random $k_w, k'_w \ \forall w \in W$.
2. For each $g \in G$ with input wires $w_0, w_1$ and output wire $w_2$, set $e_g$ as
   - $\mathsf{Enc}_{k_{w_0}}(\mathsf{Enc}_{k_{w_1}}(0^\kappa || k_{w_2}))$
   - $\mathsf{Enc}_{k_{w_0}}(\mathsf{Enc}_{k'_{w_1}}(0^\kappa || k_{w_2}))$
   - $\mathsf{Enc}_{k'_{w_0}}(\mathsf{Enc}_{k_{w_1}}(0^\kappa || k_{w_2}))$
   - $\mathsf{Enc}_{k'_{w_0}}(\mathsf{Enc}_{k'_{w_1}}(0^\kappa || k_{w_2}))$
3. Output $\widetilde{C} := ((e_g)_{g\in G}, k_{w_{\mathsf{out}}} \to C(x), k'_{w_{\mathsf{out}}} \to 1 - C(x))$.

**Figure 2:** Simulator for security of garbled circuits

We prove that the output of this simulator is indistinguishable from the actual view of the circuit evaluator via a series of hybrids:

$$H_0 := \left\{\widetilde{C}, \mathsf{lab}_{i,x_i}\right\} \stackrel{c}{=} H_1 \stackrel{c}{=} \cdots \stackrel{c}{=} H_{T-1} \stackrel{c}{=} \{\mathsf{Sim}(1^\kappa, C(x))\} := H_T$$

We proceed by replacing wires gate by gate. The idea is to replace the three non-opened entries with encryptions of the only wire which is correct. Say the input labels are $k^0_{w_0}, k^1_{w_1}$. These input labels are obtained from the simulator for the oblivious transfer. Originally, $e_g$ consists of

$$\begin{gathered}\mathsf{Enc}_{k^0_{w_0}}(\mathsf{Enc}_{k^0_{w_1}}(0^\kappa || k^{g(0,0)}_{w_2})) \\ \mathsf{Enc}_{k^0_{w_0}}(\mathsf{Enc}_{k^1_{w_1}}(0^\kappa || k^{g(0,1)}_{w_2})) \\ \vdots\end{gathered}$$

For each $a, b \in \{0,1\}$, we replace the encrypted values with $\mathsf{Enc}_{k^a_{w_0}}(\mathsf{Enc}_{k^b_{w_1}}(0^\kappa || k^{g(0,1)}_{w_2}))$. By security of the encryption scheme, this new $e'_g$ is indistinguishable from the original $e_g$. Thus each hybrid is indistinguishable from the previous one. This hybrid argument provides security for $P_1$'s input. Security of the 1-out-of-2 oblivious transfer provides privacy for $P_2$'s input. ■

# 3 GMW Protocol

Yao's protocol is limited to two party computation. Goldreich, Micali and Wigderson (GMW) created the first secure multiparty computation protocol. Here we give an informal sketch of the protocol, limiting ourselves to two parties for simplicity of exposition.

### 3.1 Construction

Let $P_1$'s input be $x_1$, and $P_2$'s input be $x_2$. Each party creates a secret share of their input and sends it to the other party. For example, $P_1$ samples a random $a$, and computes $b_1 = a_1 \oplus x_1$. $P_2$ does the same for $x_2$. $P_1$ gets the $a$ shares, and $P_2$ gets the $b$ shares. Computing $\mathsf{NOT}, \mathsf{XOR}$, and $\mathsf{AND}$ gates is done as follows:

- $\mathsf{NOT}$ gates: Each party simply flips their share of the input bit.
- $\mathsf{XOR}$ gates: Since $x_1 \oplus x_2 = (a_1 \oplus b_1) \oplus (a_2 \oplus b_2) = (a_1 \oplus a_2) \oplus (b_1 \oplus b_2)$, each party can simply XOR their shares separately.
- $\mathsf{AND}$ gates: $(a_1 \oplus b_1) \cdot (a_2 \oplus b_2) = ((a_1 \cdot a_2) \oplus (b_1 \cdot b_2)) \oplus ((a_1 \cdot b_2) \oplus (a_2 \cdot b_1))$. We see that each party can compute the first parts with the shares they possess. However, to get the remaining, they need to know the other party's shares, which compromises security. So instead, the parties utilize 1-out-of-4 oblivious transfer to compute the $\mathsf{AND}$ gate.

  The setup is as follows: $P_1$ samples a random bit $a$. This its share of the result of the gate. It computes the following table:

| $b_1$ | $b_2$ | $b_3$ |
|---|---|---|
| 0 | 0 | $(a_1 \cdot a_2) \oplus a$ |
| 0 | 1 | $(a_1 \cdot \neg a_2) \oplus a$ |
| 1 | 0 | $(\neg a_1 \cdot a_2) \oplus a$ |
| 1 | 1 | $(\neg a_1 \cdot \neg a_2) \oplus a$ |

  The intuition for the entries in the table is that when $b_1 = b_2 = 1$, the $\mathsf{AND}$ gate becomes $(a_1 \oplus b_1) \cdot (a_2 \oplus b_2) = (a_1 \oplus 1) \cdot (a_2 \oplus 1) = (\neg a_1) \cdot (\neg a_2)$. The other entries are computed similarly.

  $P_1$ computes each of the possible values of $b_3$, and feeds them as input to the OT. $P_2$ feeds in $(b_1, b_2)$ to the OT, and gets some secret share. Thus both parties possess a share of the correct output.

### 3.2 1-out-of-4 Oblivious Transfer

We describe how to implement a 1-out-of-4 OT using 1-out-of-2 OT:

1. The sender, $P_1$ samples 5 random values $S_i \leftarrow \{0,1\}, i \in \{1, \ldots, 5\}$.
2. $P_1$ computes

$$\begin{aligned}
\alpha_0 &= S_0 \oplus S_2 \oplus m_0 \\
\alpha_1 &= S_0 \oplus S_3 \oplus m_1 \\
\alpha_2 &= S_1 \oplus S_4 \oplus m_2 \\
\alpha_3 &= S_1 \oplus S_5 \oplus m_3
\end{aligned}$$

   It sends these values to $P_2$.

3. The parties engage in 3 1-out-of-2 Oblivious Transfer protocols for the following messages: $(S_0, S_1)$, $(S_2, S_3)$, $(S_4, S_5)$. THe receiver's input for the first OT is the first choice bit, and for the second and third ones is the second choice bit.

4. The receiver can only decrypt one ciphertext.