CPSC 467: Cryptography and Security

Michael J. Fischer

Lecture 24a
December 1, 2020

Secure Multiparty Computation (MPC)

Cooperation, trust, and privacy

People often desire to cooperate with others whom they don't fully trust in order to achieve a common benefit.

The model here is that each player has some private information that is needed to carry out some transaction, but they don't want the other party to learn any more of their private information that is needed to compute the desired result

The millionaires' problem

The *Millionaires' Problem*, introduced by Andy Yao in 1982, began the study of *privacy-preserving multiparty computation*.

Alice and Bob want to know who is the richer without revealing how much they are actually worth.

Alice is worth I million dollars; Bob is worth J million dollars.

They want to determine whether or not $I \geq J$, but at the end of the protocol, neither should have learned any more about the other person's wealth than is implied by the truth value of the predicate $I \geq J$.

Privacy-preserving vote counting

Another example is vote-counting.

Each voter has an input $v_i \in \{0, 1\}$ indicating their no/yes vote on a referendum.

The goal is to collectively compute $\sum v_i$ while maintaining the privacy of the individual v_i.

Further applications

- Private Auction
  - Inputs: bids
  - Outputs: winning party and winning price
- Anonymous credentials
  - Inputs: credential with personal information, e.g., driver's license
  - Output: proof of a certain property of the credentials, e.g. age over 21
- Disease tests with DNA sequences
  - Inputs: test algorithm, sequenced genome
  - Output: test result
- Data sharing between hospitals
  - Inputs: hospital databases information, e.g. driver's license
  - Output: common patients

General framework

Consider n parties $P_1, P_2, \ldots, P_n$ with private inputs $x_1, x_2, \ldots, x_n$ and a public function f.

The MPC protocol must output $f(x_1, x_2, \ldots, x_n)$ while preserving certain *security properties*, even if some of the parties collude and maliciously attack the protocol

Normally, this is modeled by an external adversary $\mathcal{A}$ that may corrupt some parties and coordinates their actions.

Some commonly studied security properties

- *Correctness*: parties obtain correct output (even if some parties misbehave)
- *Privacy*: only the output is learned (nothing else)
- *Independence of inputs*: parties cannot choose their inputs as a function of other parties' inputs
- *Fairness*: if one party learns the output, then all parties learn the output
- ...

The adversary in an MPC protocol

One could make various assumptions about the power of the adversary. Generally we model the adversary to have control over the data and actions of a set of t parties (also known as corrupt parties).

Most commonly studied adversarial models:

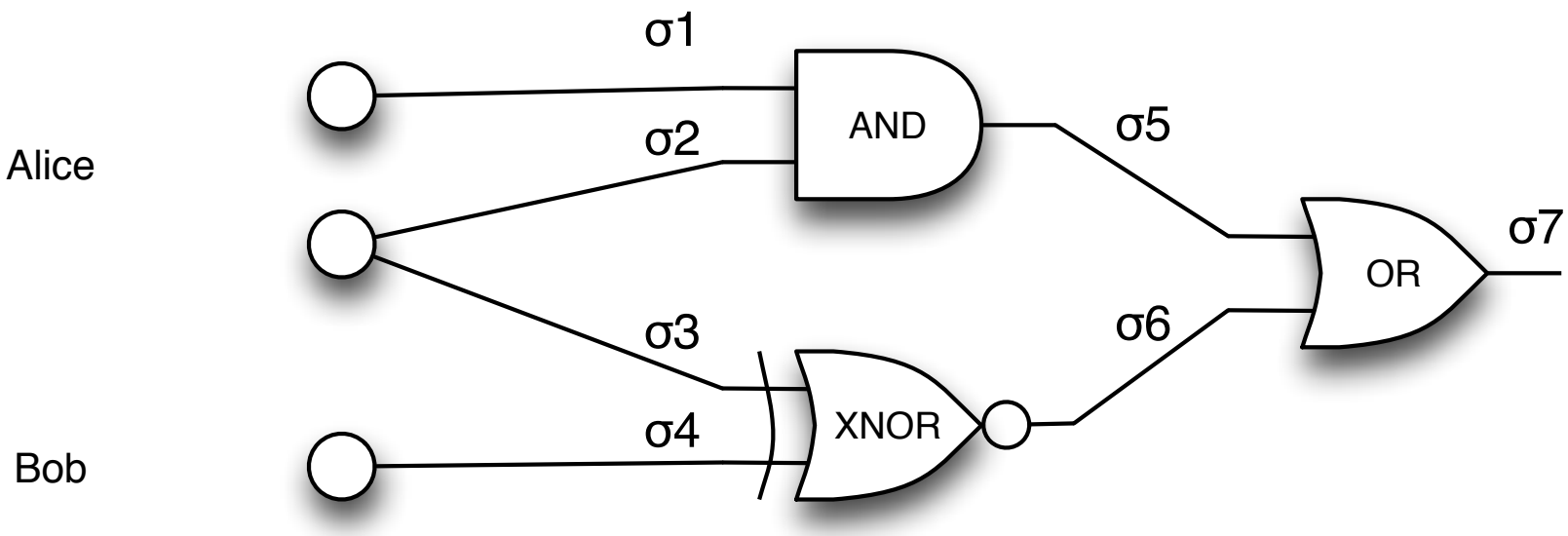
- Semi-honest adversarial model - adversary only observes the *views* of the corrupt parties
- Malicious adversarial model - adversary controls the behavior of the corrupt parties during the protocol

Private Boolean Circuit Evaluation

Boolean functions computed by circuits

Let $\bar{z} = f(\bar{x}, \bar{y})$, where $\bar{x}$, $\bar{y}$, and $\bar{z}$ are bit strings of lengths n_x, n_y, and n_z, respectively, and f is a Boolean function computed by a polynomial size Boolean circuit C_f with $n_x + n_y$ input wires and n_z output wires.

Example:



Private evaluation of a functionality

In a *private evaluation* of C_f, Alice furnishes the (private) input data to the first n_x input wires, and Bob furnishes the (private) input data for the remaining n_y input wires. The n_z output wires should contain the result $\bar{z} = f(\bar{x}, \bar{y})$. The corresponding *functionality* is

$$F(\bar{x}, \bar{y}) = (\bar{z}, \bar{z}).$$

Alice and Bob should learn nothing about each other's inputs or the intermediate values of the circuit, other than what is implied by their own inputs and the output values $\bar{z}$.

Circuit evaluation

An evaluation of a circuit assigns a Boolean value σ_w to each wire of the circuit. The input wires are assigned the corresponding input values.

Let G be a gate with input wires u and v and output wire w that computes the Boolean function $g(x, y)$. In a correct assignment, $\sigma_w = g(\sigma_u, \sigma_v)$.

A complete evaluation of the circuit first assigns values to the input wires and then works its way down the circuit, assigning a value to the output wire of any gate whose inputs have already received values.

Private circuit evaluation

In a *private circuit evaluation*,

- Both Alice and Bob learn the output values of the circuit;
- Neither Alice nor Bob gain any information about each other's private input values except for whatever is implied by their own input values and the circuit output.

Many different schemes exist for privately evaluating a circuit. We will be discussing one of the conceptually simplest schemes based on *value shares*.

Secure Circuit Evaluation Using Value Shares

Value shares

In a private evaluation using *value shares*, we split each value σ_w into two random shares a_w and b_w such that $\sigma_w = a_w \oplus b_w$.

- Alice knows a_w; Bob knows b_w.
- Neither share alone gives any information about σ_w, but together they allow σ_w to be computed.

After all shares have been computed for all wires, Alice and Bob exchange their shares a_w and b_w for each output wire w.

They are both then able to compute the circuit output.

Obtaining the shares

We now describe how Alice and Bob obtain their shares while maintaining the desired privacy.

There are three cases, depending on whether w is

1. An input wire controlled by Alice;
2. An input wire controlled by Bob;
3. The output wire of a gate G.

Alice's input wires

1. Input wire controlled by Alice:

   Alice knows σ_w.

   She generates a random share $a_w \in \{0, 1\}$ for herself and sends Bob his share $b_w = a_w \oplus \sigma_w$.

Bob's input wires

2. Input wire controlled by Bob:

   Bob knows σ_w.

   Alice chooses a random share $a_w \in \{0, 1\}$ for herself.

   She prepares a table T:

σ	$T[\sigma]$
0	a_w
1	$a_w \oplus 1$.

   Bob requests $T[\sigma_w]$ from Alice via OT_1^2 and takes his share to be $b_w = T[\sigma_w] = a_w \oplus \sigma_w$.

Obtaining shares for gate output wires

3. Output wire of a gate G:
   Let G have input wires u, v and compute function $g(x, y)$.
   Alice chooses random share $a_w \in \{0, 1\}$ for herself.
   She computes the table

$$\begin{aligned} T[0,0] &= a_w \oplus g(a_u, a_v) \\ T[0,1] &= a_w \oplus g(a_u, a_v \oplus 1) \\ T[1,0] &= a_w \oplus g(a_u \oplus 1, a_v) \\ T[1,1] &= a_w \oplus g(a_u \oplus 1, a_v \oplus 1) \end{aligned}$$

   (Equivalently, $T[r, s] = a_w \oplus g(a_u \oplus r, a_v \oplus s)$.)

   Bob requests $T[b_u, b_v]$ from Alice via OT_1^4 and takes his share to be $b_w = T[b_u, b_v] = a_w \oplus g(\sigma_u, \sigma_v)$.

Remarks

1. Alice and Bob's shares for w are both independent of σ_w.
   - Alice's share is chosen uniformly at random.
   - Bob's share is always the XOR of Alice's random bit a_w with something independent of a_w.
2. This protocol requires n_y executions of OT_1^2 to distribute the shares for Bob's inputs, and one OT_1^4 for each gate.[1]
3. This protocol assumes semi-honest parties.
4. Bob does not even need to know what function each gate G computes. He only uses his private inputs or shares to request the right line of the table in each of the several OT protocols.

[1] Note: The n_y executions of OT_1^2 can be eliminated by having Bob produce the shares for his input wires just as Alice does for hers. Our approach has the advantage of being more uniform since Alice is in charge of distributing the shares for all wires.