Today: Information theoretic secure multi-party computation (MPC) [BenOr-Goldwasser-Wigderson 88]

Previous 4 lectures: Secure MPC assuming OT.
- 2-party honest-but-curious (HBC):
  Yao Garbled circuits + HBC OT
- Multi-party HBC:
  secret-share of inputs + (gate-by-gate comp) + reconstruction, using HBC OT
- Multi-party malicious security:
  Add commitments + ZK proofs-of-knowledge + coin tossing

Today: MPC without any computational assumptions.
HBC setting.
(Next lecture we will talk briefly about malicious setting).

Thm [BGW]: For any (n-input) function \( f \), \( \exists \) MPC protocol for \( f \) with all powerful perfectly secure against HBC adv. that corrupts
at most \( t < \frac{n}{2} \) parties, assuming each pair of parties is connected via a private & authenticated communication channel.

(Such assumption is necessary in order to obtain perfect security).

**Def:** Let \( f : (\{0, 1\}^*)^n \rightarrow (\{0, 1\}^*)^n \) be a function.

An \( n \)-party protocol \( \Pi \) \( t \)-privately computes \( f \) in the HBC setting if \( \forall x = (x_1, \ldots, x_n) \) s.t. \( |x_1| = \ldots = |x_n| \)

\[ \text{Output } \Pi(x_1, \ldots, x_n) = f(x_1, \ldots, x_n) \]

& \exists \text{ PPT } S \text{ s.t. } \forall I \subseteq [n] \text{ of size } \leq t \quad \forall x = (x_1, \ldots, x_n) \text{ s.t. } |x_1| = \ldots = |x_n| \]

\[ S(I, (x_i)_{i \in I}, (f_i(x))_{i \in I}) = \text{View}_I \Pi(x) \]

the view of all parties \( i \in I \).

this view consists of their input, randomness, and all the msgs they received during the protocol.

\((t\text{-private } = \text{ perfect security against } t \text{ corruptions})\)
The BGW Protocol: HBC setting

The blueprint of the BGW protocol is similar to that of the GMW protocol. It consists of 3 phases:

1. Secret sharing of inputs
2. Gate-by-gate computation
3. Output reconstruction.

The way these phases are implemented in the BGW is very different than the way they are implemented in GMW.

In particular, in the information theoretic setting there is no generic transformation that converts HBC private protocol to malicious private one.

(We cannot use ZK proofs...)

The HBC setting itself is also implemented quite differently. The inputs are not shared via XOR. Rather:

A central tool in the BGW protocol is Shamir's secret sharing scheme.
Shamir's Secret Sharing Scheme

Loosely speaking, a t-out-of-n secret sharing scheme takes as input a secret $s$ (from some domain) and outputs $n$ shares, with the property that it is possible to efficiently reconstruct $s$ from every subset of $t$ shares, but every subset of less than $t$ shares reveals nothing about the secret $s$.

The value $t$ is called the threshold of the scheme.

**Def:** A t-out-of-n secret sharing scheme consists of 2 PPT alg: (Share, Reconstruct).

Share takes as input a secret $s$ (from some domain) and outputs $n$ shares $(b_1,\ldots,b_n)$.

Reconstruct takes as input $t$ shares $(b_{i_1},\ldots,b_{i_t})$ (together with the corresponding indices $(i_1,\ldots,i_t)$) and outputs a secret $s$.

The following two properties are guaranteed to hold:

**Correctness:** \( \forall S \ \forall I \subseteq [n] \ |I| = t \)

\[
Pr[\text{Reconstruct}(b_{i_1}, i) = S] = 1
\]

where the prob is over $(b_1,\ldots,b_n) \leftarrow \text{Share}(S)$,
and where \( B_i = (B_i^j)_{i \in I} \)

Security: \( \forall S, S' \) (in the domain) \( \forall I \subset [n] \) \( |I| < t \)

\[ B_i = B_i^I \]

where \( (B_1, ..., B_n) \leftarrow \text{Share}(S) \) & \( (B_1^t, ..., B_n^t) \leftarrow \text{Share}(S') \).

Shamir's Construction: \( t \)-out-of-\( n \)

Let \( F \) be a finite field of size \( |F| > n \). Fix \( a_1, ..., a_n \in F \) non-zero & distinct.

\text{Share}(S): \ Choose \ at \ random \ degree \ t-1 \ polynomial

\[ g(x) \in F[x] \text{ s.t. } g(0) = S. \] \ This \ is \ done \ by \ choosing

\text{at} \ random \ \( c_1, ..., c_{t-1} \in F \) \text{ and setting}

\[ g(x) = \sum_{i=1}^{t-1} c_i x^i + S. \] \ Let \( \beta_i = g(a_i) \)

\text{output} \ (\beta_1, ..., \beta_n)

\text{Reconstruct} \ ((\alpha_i, \beta_i), ..., (\alpha_t, \beta_t)) \ finds \ the \ unique

polynomial \( g(x) \) of degree \( t-1 \) st. \( \forall j \in [t] \) \( g(\alpha_j) = \beta_j \),

and outputs \( S = g(0) \).

This is done via Lagrange interpolation, as follows:
\[ g'(x) = \sum_{i \in I} \beta_i \lambda_{\alpha_i}(x), \text{ where} \]
\[ \lambda_{\alpha_i}(x) = \prod_{j \in \{1, \ldots, i-1\}} \frac{x - \alpha_j}{\alpha_i - \alpha_j} \]

Note that \( \lambda_{\alpha_i}(\alpha_i) = 1 \) and \( \forall j \in I \setminus \{i\} \lambda_{\alpha_i}(\alpha_j) = 0 \)

\[ \Rightarrow g'(\alpha_i) = \beta_i \quad \forall i \in I. \quad \text{The fact that } g = g' \text{ follows from the following claim.} \]

\textbf{Claim:} For any \( t \) pairs \( \{(\alpha_i, \beta_i)\}_{i=1}^{t} \), where \( \alpha_1, \ldots, \alpha_t \) are all distinct, there is a unique degree \( t-1 \) polynomial \( g \) s.t. \( g(\alpha_i) = \beta_i \) (and it is the one computed above).

\textbf{Proof:} Assume for contradiction that there exist distinct \( g, g' \) both polynomial of degree \( t-1 \), s.t.
\[ g(\alpha_i) = g'(\alpha_i) = \beta_i \quad \forall i \in [t]. \]

\[ \Rightarrow P = g - g' \text{ is a degree } t-1 \text{ polynomial with at least } t \text{ roots } \alpha_1, \ldots, \alpha_t \]

\[ \Rightarrow P = 0 \Rightarrow g = g' \text{, contradiction.} \]

Relies on the following claim:
Claim: Every non-zero polynomial of deg $d$ has at most $d$ roots.

Proof: By induction. For $d=1$, $g=ax+b$  
Suppose the claim is true for $d-1$, we will prove it for $d$.
Take any non-zero poly $g(x)$ of deg $d$.
If it has a root $\alpha$ then $(\alpha-\alpha)|g(x) = 0$.

$g(x) = g_1(x)(x-\alpha)$

deg $d-1$ and thus has at most $d-1$ roots by induction.

Note: Reconstruction can also be done using the Vandermonde matrix: Given $\alpha_1, \ldots, \alpha_d \in F$

$$V_{\alpha_1, \ldots, \alpha_d} = \begin{pmatrix}
1 & \alpha_1 & \alpha_1^2 & \cdots & \alpha_1^{d-1} \\
1 & \alpha_2 & \alpha_2^2 & \cdots & \alpha_2^{d-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & \alpha_d & \alpha_d^2 & \cdots & \alpha_d^{d-1}
\end{pmatrix}$$

Fact: $V_{\alpha_1, \ldots, \alpha_d}$ is invertible iff $\alpha_1, \ldots, \alpha_d$ are all distinct & non-zero.
We can find \( q = \sum_{i=0}^{t-1} c_i x^i \) s.t. \( q(x_i) = \beta_i \).

as follows:

\[
V_{x_1 \ldots x_t} \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{t-1} \end{pmatrix} = \begin{pmatrix} \beta_1 \\ \vdots \\ b_t \end{pmatrix}
\]

\( \Rightarrow \begin{pmatrix} c_0 \\ c_1 \\ \vdots \\ c_{t-1} \end{pmatrix} = V_{x_1 \ldots x_t}^{-1} \begin{pmatrix} \beta_1 \\ \vdots \\ b_t \end{pmatrix} \)

Output \( c_0 \) as the secret.

Claim: The Shamir Secret Sharing scheme is secure.

(Follows from basic linear algebra).

The **BGW Protocol** (HBC)

Let \( f \) be a function that the parties wish to compute.

Let \( C \) be an arithmetic circuit that computes \( f \).

We assume (without loss of generality) that the arithmetic operations in \( C \) are over a large field \( \text{IF} \) of size \( > n \).
This is WLOG since one can convert binary operations over $GF(2)$ to be over $GF$ by converting $b_1 + b_2$ to $b_1 + b_2 - 2b_1 b_2$ and keeping $b_1 b_2$ as is.

We also assume for simplicity that the input of each party is a single element in $F$.

The protocol works by having the parties jointly compute the circuit from the input gates to the output gate, keeping the invariant that at each stage of the comp. the parties hold Shamir Shares of the values of all the wires of the gates that have been computed this far. (So, in some sense, its shares are distributed like fresh Shamir shares.) Specifically, the protocol consists of 3 phases:

- **Input stage**: Each party creates shares of its input using Shamir's secret sharing scheme with threshold $t + 1$ (for a given $t < \frac{n}{2}$), and distributes the shares among the parties.

Namely, each party $i$, given input $x_i \in \mathbb{F}$, generates
a random degree t polynomial \( g_i \) s.t. \( g_i(0) = x_i \).

For \( j \in [n] \), \( P_i \) sends \( P_j \) the share \( g_i(\alpha_j) \).

- **Computation stage**: In this stage the parties jointly compute the circuit \( C \), gate-by-gate, as follows:

  - **Addition gates**: Given shares to the input wires of the gate the shares for the output wire are generated by adding the input shares, locally. No interaction is needed.

  More formally, suppose the two inputs to the gates are \( a, b \), and the parties' shares are given by the degree \( t \) polynomials \( f_a \) & \( f_b \).

  Then party \( P_i \), given shares \( f_a(\alpha_i), f_b(\alpha_i) \), computes the share of the output wire by setting

  \[
  f_{a+b}(\alpha) = f_a(\alpha_i) + f_b(\alpha_i) = (f_a + f_b)(\alpha_i)
  \]

  deg \( t \) poly w. free coefficient \( a+b \), as desired. By induction, distributed like a random deg \( t \) poly w. free coefficient \( a+b \).
Multiplication gate: As before, let \( f_a \) & \( f_b \) denote the polynomials defining the shares of the two input wires of the gate, where \( a \) & \( b \) are the values of these input wires.

As in the addition case, each party can simply compute \( f_a(x_i) \cdot f_b(x_i) \approx h(x_i) \).

The free coefficient of \( h \) is indeed \( a \cdot b \).

However, \( h \) may be of degree \( 2t \), instead of \( \deg t \).

Furthermore, \( h \) is not a random polynomial, but rather has a specific structure (since it is a product of \( 2 \cdot \deg t \) poly's).

Instead, the parties will run an interactive protocol that computes the (randomized) function \( F_{\text{mult}} \):

\[
F_{\text{mult}}((f_a(x_1), f_b(x_1)), \ldots, (f_a(x_n), f_b(x_n))) =
(f_{a \cdot b}(x_1), \ldots, f_{a \cdot b}(x_n))
\]

where \( f_{a \cdot b} \) is a random degree \( t \) polynomial with free coefficient \( a \cdot b \).

(More details in a bit...)
Output stage: At the end of the computation stage, the parties hold shares of the output wires. The parties simply send their shares to the appropriate party.

Namely, if there are \( n \) output wires \( o_1, \ldots, o_n \), where \( o_i \) is the output of party \( i \).

Denote by \( \beta_i, \ldots, \beta_k \) the shares of wire \( o_k \).

Then each party \( P_i \) sends \( \beta_{i,k} \) to party \( P_k \) (\( k \)).

Upon receiving all shares, party \( P_i \) uses \( \beta_i \) to reconstruct a degree \( t \) poly \( g \) s.t. \( g(a_i) = \beta_i \) and outputs \( g(0) \).

Thm: This protocol is \( t \)-private against \( t \)-BCC adversaries in the \( \text{F}_{\text{mult}} \)-hybrid model assuming we had a perfect implementation of \( \text{F}_{\text{mult}} \).

Proof idea: One can argue by induction that the shares for each gate are close to a random degree \( t \) poly \( w \) s.t. the coefficient being the value of the gate, where the shares of an addition gate are simply
the sum of the shares of the input wires.

Thus the simulator can simulate these shares by choosing random shares for input wires of honest parties, and honestly choosing shares on behalf of corrupt parties. Then for addition gate simulate the shares being the addition of the input shares, and for mult gate simulate the shares to be random.

For the output gate corresponding to an adv. party, the simulator, given t shares corresponding to the output wire will compute the (unique) degree t ply g that agrees with these shares, and s.t. \( g(0) = f(x_1, x_2) \)

and simulate the shares as being \( \{ g(x_i) \}_{i=1}^n \).

\[ \text{Note: So far we did not use the fact that } t < \frac{n}{2}. \]

This will be used to privately compute \( F_{\text{mult}} \).

The "proof sketch" above works if t!
Privately computing $F_{\text{mult}}$

Recall:

$$F_{\text{mult}}\left((f_a(x), f_b(x)), \ldots, (f_a(x_n), f_b(x_n))\right) = \left( f_{ab}(x_1), \ldots, f_{ab}(x_n) \right)$$

where $f_{ab}$ is a random degree $t$ poly s.t. $f_{ab}(0) = a \cdot b$.

As we mentioned, the simple solution of simply multiplying the shares doesn't work for two reasons:

1. The resulting poly is of deg $2t$.
2. """" is not random, but rather is structured (i.e., a product of $2$ deg. $t$ polys).

Instead $F_{\text{mult}}$ is computed as follows:

1. Each party locally multiplies its input shares:
   
   Each party $P_i$, given input shares $\beta_i \text{ and } \delta_i$, computes
   
   $$s_i = \beta_i \cdot \delta_i$$

2. Randomize: The parties run a protocol that randomizes the above deg $2t$ polynomial, without changing the free coefficient:
   
   Each party $P_i$ chooses a random deg $2t$ poly $g_i$ s.t. $g_i(0) = 0$, and sends $g_i(x_i)$ to each party $P_k$. 
Each party $P_i$, upon receiving shares $\{G_i^k\}_{i \in [n]}$, sets $S_i = S_i + \sum_{i=1}^n G_i^k$

These shares correspond to the polynomial
\[
h = f_a \cdot f_b + \frac{1}{n} \sum_{i=1}^n g_i \quad \text{which is a randomdeg2t poly st. } h(0) = a \cdot b
\]

3. Degree reduction: The parties run a protocol to reduce the degree of the shared poly from $2t$ to $t$.
   This is where the bound $t < n/2$ is used!

   Let $h(x) = \sum_{i=0}^{2t} h[i] x^i$ be a deg $2t$ poly, and denote by $\text{trunc}_t(h(x)) = \sum_{i=0}^t h[i] x^i$.

   In this step the parties, given inputs $\beta_1, \ldots, \beta_n$, compute
   \[
   (\delta_1, \ldots, \delta_n) = F_{\text{deg-reduce}}(\beta_1, \ldots, \beta_n)
   \]
   where
   \[
   \text{reconstruct}(\delta_1, \ldots, \delta_n) = \text{trunc}_t(\text{reconstruct}(\beta_1, \ldots, \beta_n)).
   \]

   Note that since reconstruct($\beta_1, \ldots, \beta_n$) is a random poly, it follows that reconstruct($\delta_1, \ldots, \delta_n$) is also a random poly, since it has the same coefficients.
We first show that in order to convert the shares $(\beta_1, \ldots, \beta_n)$ of the polynomial $h$ to shares of the poly
trunc$_t(h)$, it suffices to multiply the input shares by a fixed matrix $A$, which we define below.

Denote $h(x) = \sum_{i=0}^{2t} h[i] x^i$, and denote

$$\overrightarrow{h} = (h[0], \ldots, h[2t], 0, \ldots, 0) \in \mathbb{F}^n.$$  

By def $\beta_i = h(\alpha_i) \quad \forall i \in [n]$.

Let $V_{\alpha} = V(\alpha_1, \ldots, \alpha_n)$ be the $n \times n$ Vandermonde matrix:

$$V_{\alpha} = \begin{pmatrix} 1 & \alpha_1 & \cdots & \alpha_1^{n-1} \\ 1 & \alpha_2 & \cdots & \alpha_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha_n & \cdots & \alpha_n^{n-1} \end{pmatrix}$$

As we have seen:

$$V_{\alpha} \cdot (\overrightarrow{h})^t = (\overrightarrow{\beta})^t \quad \text{assuming } 2t \leq n$$

Since $V_{\alpha}$ is invertible

$$(\overrightarrow{h})^t = V_{\alpha}^{-1} (\overrightarrow{\beta})^t$$
Let \( g = \text{trunc}_t(h) \) be the truncated poly

\[
g(x) = \sum_{i=0}^{t} h[i] x^i.
\]

By def, \( \delta_i = g(x_i) \). Furthermore, denoting

\[
\overline{g} = (h[0], \ldots, h[t], 0, \ldots, 0) \in \mathbb{F}^n,
\]

we have that

\[
V_x \cdot (\overline{g})^t = (\overline{\delta})^t.
\]

Let \( T = \{1, \ldots, t^3\} \), and let \( P_T \) be the linear projection of \( T \), i.e.

\[
P_T = \begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 0
\end{pmatrix}
\in \mathbb{F}^{n \times n}
\]

\[
P_T(i, i) = 1 \text{ if } i \in T
\]

\[
P_T(i, j) = 0 \text{ o.w.}
\]

So

\[
P_T(\overline{h})^t = (\overline{g})^t
\]

\[
V_x \cdot P_T \cdot (\overline{h})^t = (\overline{\delta})^t
\]

\[
V_x \cdot P_T \cdot V_x^{-1} (\overline{\beta})^t = (\overline{\delta})^t
\]

Let \( A = V_x \cdot P_T \cdot V_x^{-1} \) be a fixed matrix.

Then \( A(\overline{\beta})^t = (\overline{\delta})^t \)

assuming at \( \leq n \).
Compute $A\cdot(B^t)$ using $t$-private protocol for computing linear functions!

Namely: each party $P_i$ shares $B_i$ via random deg $t$ poly $f_i$ s.t. $f_i(0) = B_i$. Then the parties compute mul. by scalar by simply locally mul. by the scalar, and compute addition by locally adding the shares.

The Malicious Setting

**Thm [BGW]**: For all $n$-input function $f$, $\exists$ MPC protocol for $f$ with perfect security against a malicious adv.

that corrupts at most $t < \frac{1}{3}$ players

(assuming each pair of parties are connected via a private & authenticated communication channel).

**Def**: Let $f : (\{0,1\}^n)^n \to (\{0,1\}^n)^n$ be a function.

An $n$-party protocol $\Pi$ $t$-privately computes $f$ in the malicious setting if $\forall$ prob. adv $A$ (no runtime restriction)

$\exists$ prob. simulator $S$ of "comparable complexity" s.t.
\[ \forall i \in [n] \quad \forall x = (x_1, \ldots, x_n) \quad |x_1| = \ldots = |x_n| \quad \forall \alpha \in [0, \beta', 1] \]

\[ \text{Ideal } f, S(z(x_i), I) = \text{Real } \pi, A(z, x_i), I \quad \text{view of } A \text{ in the real world.} \]

with fairness

```
\[ f \quad \text{ sends output to all parties} \]
```

\[ x_i \quad \text{corrupted party } P_i \]

\[ x_i \quad \text{honest party } P_i \]

**Note:** The previous protocol is not secure against malicious adv.

Recall in Shamir's secret sharing scheme, the party sharing its input (refereed as the dealer) is supposed to generate a random deg t poly \( g \) s.t. \( g(0) = \text{input} \).

What if this party is adv, and instead chooses \( g \) to be of high degree?

Or what if an adv party who received a share \( \beta_i \) pretends that it received a share \( \beta_i' \)? All goes wrong
Instead to get malicious security, we use a different primitive called:

**Verifiable Secret Sharing (VSS)**

VSS is a protocol for sharing a secret in the presence of malicious adversaries.

The BGW VSS protocol solves the above problems by adding elements to the share stage that restrict the dealer to send the honest parties shares $g(x_i)$ where $g$ is a degree $t$ polynomial (even if the dealer is malicious).

Then they show that as long as $t < \frac{n}{3}$ it is possible to use techniques from the field of error correcting codes to reconstruct the polynomial $g$, as long as at least $n-t$ of the shares are honest.

More specifically, they observe that Shamir's secret sharing scheme is exactly a Reed-Solomon codes, which can correct up to $t < \frac{n}{3}$ errors.