Privacy Enhancing Technologies FS2025
Lecture 17-18-19 – Secure Multi-Party Computation

Florian Tramèr

AGENDA

1. Defining MPC
2. Computing on Secret Shared Data
3. Malicious MPC
4. Secret Sharing

Recap

We have seen a lot of different cryptographic primitives in this course, that apply for specific problems:

- Commitment schemes
- Zero-knowledge proofs
- PIR
- ORAM
- ...

All of these primitives can be seen as a way for distrusting parties to compute some function across their respective data, without leaking information to others. Today, we will see how to build a more general protocol that allows multiple parties to compute an arbitrary function over their data, without leaking any excess information to each other.

1 Secure Multi-Party Computation

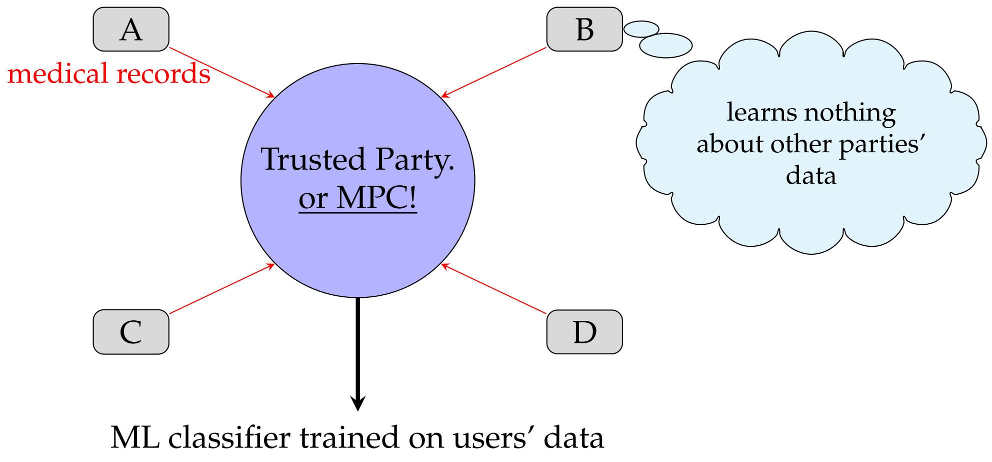
Today's lecture will be devoted to one of the most fascinating result of modern cryptography:

> *"Any function that can be computed securely with the help of a trusted third party, can also be securely computed without."*

We will unpack what this statement means formally in the following section. But for now, let's look at some examples.

Example application: Private Machine Learning. Suppose that you have different parties that want to jointly train a machine learning model on their data. These could be different hospitals that want to combine their local patient datasets to train a better model for predicting diseases. Or it could be all smartphone owners jointly training a keyboard prediction model on all of their text messages. In both cases, the parties want to obtain a good model by combining everyone's data, but they do not want to reveal their data to each other.

The "trivial" solution is to have a third party that collects everyone's data, trains the model, and then sends it back to the parties. But of course this is not very satisfying, since it requires entrusting one party with everyone's data. Secure multi-party computation (MPC) basically allows us to run a protocol that "simulates" the trusted third party, but without actually having to trust anyone!


A
medical records
B
learns nothing about other parties' data
Trusted Party. or MPC!
C
D
ML classifier trained on users' data

Other applications. Pretty much any cryptographic application you can think of can be cast as a secure multi-party computation problem. Examples include:

- Secure messaging
- E-voting
- Private auctions
- ...

So why don't we use MPC for *everything, everywhere, all at once?* Well, its generality comes at a cost: MPC is typically very inefficient compared to a non-private solution (e.g., the overhead for training ML models is likely 100-1000×). And there are also specialized cryptographic protocols that are much more efficient for specific applications (e.g., commitments, ZK proofs, etc.). But MPC has made enormous practical progress in the last few years, so it is likely that we will see it used more and more in the future.

1.1 Defining MPC

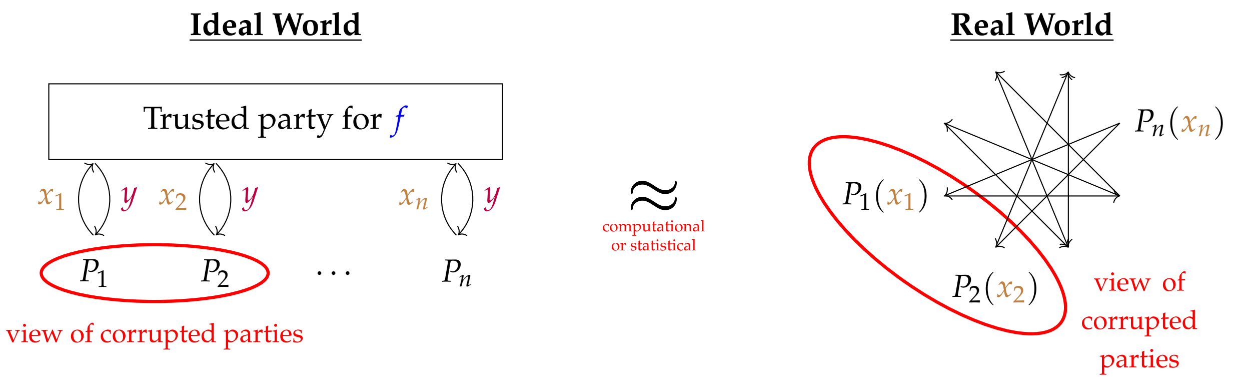
There are n parties $P_1, \dots, P_n$ with inputs $x_1, \dots, x_n$ that want to compute a function

$$y \leftarrow f(x_1, \dots, x_n)\,.$$

We can generalize this to a function that gives different outputs $y_1, \dots, y_n$ to each party.

We assume an adversary who "corrupts" some number of parties and makes them collude to break the security of the protocol.

The informal security goal is that the adversary should learn nothing more about the honest parties' inputs than it couldn't also have learned if all parties were interacting with a trusted third party.


Ideal World
Trusted party for f
x_1 y x_2 y x_n y
P_1 P_2 $\cdots$ P_n
view of corrupted parties
$\approx$
computational or statistical
Real World
$P_n(x_n)$
$P_1(x_1)$
$P_2(x_2)$
view of corrupted parties

This is sometimes called the "real ideal paradigm" or "simulation paradigm".

So what does an adversary learn in the "ideal" world?

- The inputs of the corrupted parties.
- The output y of the function f.

And that's it! In the ideal world, anything else that the adversary learns must be computable just from these two pieces of information. Thus our security goal will be that the adversary learns nothing more in the real world. For this, we will use the notion of *simulation* that we introduced for defining zero-knowledge.

Recall that in the case of zero-knowledge, we sometimes distinguished between honest verifiers and arbitrary verifiers. In the case of MPC, it is also common to distinguish between two security models:

1. *Semi-honest security.* In this model, the adversary is assumed to follow the protocol specification, but it attempts to learn anything it can about the honest parties' inputs from the protocol transcript. In particular, such an adversary can do any amount of additional (poly-time) local computation to extract information about other parties' inputs.
2. *Malicious security.* In this model, the adversary may arbitrarily deviate from the protocol specification at any time, to learn as much as possible about the honest parties' inputs or to fool them into accepting an incorrect output.

When building MPC protocols, it is often easier to first design a protocol that is secure against semi-honest adversaries, and then add additional checks and balances to prevent malicious deviations from the protocol. Semi-honest protocols are typically quite simple conceptually, and fairly efficient! We will focus on these for now, and discuss some techniques to achieve malicious security at the end of the lecture.

We can now properly formalize the security of MPC protocols in the semi-honest model.

Definition 1 (Secure (semi-honest) MPC)**.** A protocol securely computes f in the presence of a semi-honest adversary if there exists a simulator Sim such that for all inputs $x_1, \dots, x_n$, and every set of corrupt parties $C \subseteq [n]$, we have

$$\mathtt{Sim}(C, \underbrace{\{x_i : i \in C\}}_{\text{the inputs of corrupted parties}}, \underbrace{y = f(x_1, \dots, x_n)}_{\text{the output of the computation}}) \approx \underbrace{\{\mathtt{View}_i : i \in C\}}_{\text{the views of all corrupted parties in a real protocol execution}} .$$

2 The GMW protocol

The main paradigm for MPC is to have the parties *secret-share* their inputs among each other and to then compute the function f on top of secret-shared data. At the end of the computation, the parties can reconstruct the output.

We assume that each party's input is a value $x_i \in \mathbb{F}_p$, and that the function f is represented as an *arithmetic circuit* over $\mathbb{F}_p$, i.e., $y \leftarrow \mathcal{C}(x_1, \ldots, x_n)$.

Definition 2 (Additive Secret Sharing)**.** To share a secret $\alpha \in \mathbb{F}_p$ among n parties, sample random values $s_1, \ldots, s_n - 1 \leftarrow \mathbb{F}_p$ and set $s_n = \alpha - \sum_{i=1}^{n-1} s_i$. We use $[\alpha]$ to denote the additive secret share of α:

$$[\alpha] = (s_1, \ldots, s_n) \quad \text{such that} \quad \alpha = \sum_{i=1}^{n} s_i\,.$$

Lemma 1. Given an additive secret sharing of α among n parties, any subset of $n-1$ parties learns nothing about α.

Proof. Suppose the parties are $P_1, \ldots, P_n$ and the shares are $[\alpha] = (s_1, \ldots, s_n)$. Any subset of $n-1$ parties can compute the sum of their shares, which is $\alpha - s_j$ for some j. But this is just a uniformly random value in $\mathbb{F}_p$, so they learn nothing about α. □

The following MPC protocol is due to Goldreich, Micali, and Wigderson [GMW87] and is based on a simple observation:

To get a semi-honest protocol for computing f, all we need is the ability to securely compute additions and multiplications over secret-shared data!

The protocol then operates in rounds as follows:

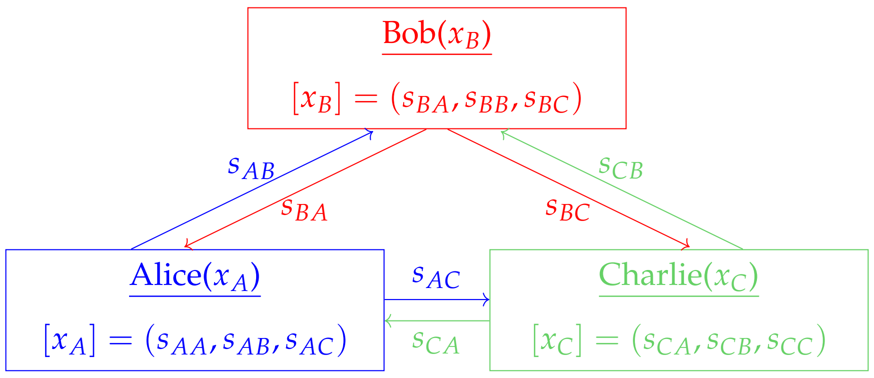
1. Each party secret shares their input with every other party.
2. For each addition gate in the circuit, with inputs $[x], [y]$, the parties run a sub-protocol to compute shares of $[x+y]$.
3. For each multiplication gate in the circuit, with inputs $[x], [y]$, the parties run a sub-protocol to compute shares of $[x \cdot y]$.
4. At the end of the protocol, each party has a secret share of the output which they reveal to all other parties.

Let's unpack this protocol step by step.

2.1 Sharing Inputs

To begin, the parties simply secret-share their inputs $x_1, \ldots, x_n$ among each other.

For example, for three parties, this would look like this:



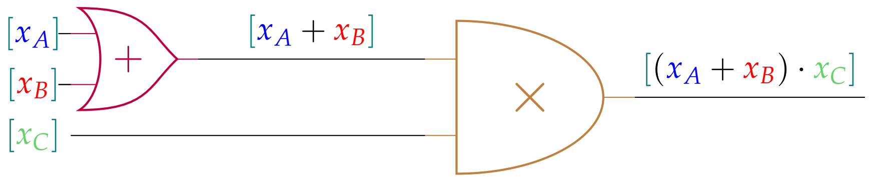
Bob(x_B)
$[x_B] = (s_{BA}, s_{BB}, s_{BC})$
s_{AB}
s_{BA}
s_{CB}
s_{BC}
Alice(x_A)
$[x_A] = (s_{AA}, s_{AB}, s_{AC})$
s_{AC}
s_{CA}
Charlie(x_C)
$[x_C] = (s_{CA}, s_{CB}, s_{CC})$

2.2 Maintaining Secret-shared Values

Throughout the protocol, the values on the wires of the circuit are always additively secret-shared. For example, suppose three parties want to compute the function

$$f(x_A, x_B, x_C) = (x_A + x_B) \cdot x_C$$

We write this down as an arithmetic circuit and run the protocol gate by gate.



Then, we use specific sub-protocols to compute secret-shared values on the wires of the circuit:

Party A's shares	s_{AA}	s_{BA}	s_{CA}	s_A^{add}	s_A^{mul}
Party B's shares	s_{AB}	s_{BB}	s_{CB}	s_B^{add}	s_B^{mul}
Party C's shares	s_{AC}	s_{BC}	s_{CC}	s_C^{add}	s_C^{mul}
Secret wire value	x_A	x_B	x_C	$x_A + x_B$	$(x_A + x_B) \cdot x_C$

2.3 Addition Gates

Addition gates are easy! Given shares of x and y, we simply have $[x + y] = [x] + [y]$ where addition of shares is done component-wise. Let's unpack this:

$$\text{If } [x] = (x_1, x_2, \ldots, x_n) \quad \text{where} \quad x = \sum_{i=1}^{n} x_i$$

$$[y] = (y_1, y_2, \ldots, y_n) \quad \text{where} \quad y = \sum_{i=1}^{n} y_i$$

$$\text{Then } [x + y] = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n) \quad \text{satisfies} \quad x + y = \sum_{i=1}^{n} (x_i + y_i)\,.$$

So processing an addition gate is easy: each party simply adds their shares of the inputs locally. This sub-protocol involves no communication between parties and provides information-theoretic security (any subset of $n - 1$ corrupt parties learns nothing about the gate's inputs or output).

2.4 Gates Involving Constants

Other operations that are easy to compute locally are addition and multiplication by a constant k:

$$\begin{aligned}[kx] &= (kx_1, kx_2, \dots, kx_n) = k[x]\,. \\ [x+k] &= (x_1 + k, x_2, \dots, x_n) \quad \text{(only party 1 adds } k \text{ to their share)}\,.\end{aligned}$$

2.5 Multiplication Gates

The trick above does *not* work for multiplication gates.

$$[x \cdot y] \neq [x] \cdot [y] = [x_1 \cdot y_1, x_2 \cdot y_2, \dots, x_n \cdot y_n] \qquad \text{that is,} \qquad x \cdot y \neq \sum_{i=1}^{n} x_i \cdot y_i\,.$$

In fact, there is no way to compute $[x \cdot y]$ locally, without some form of interaction between the parties. This is where the bulk of the work in MPC protocols goes into.

2.5.1 The Trusted Dealer and Beaver's Randomization Trick

So multiplication gates are expensive: each one of them requires a sub-protocol that involves communication between the parties.

As a first step, we will assume that there is an extra $n+1$th party, called the *trusted dealer*. This party does not have an input to the function f, and generates random values for the other parties to use. In the semi-honest setting, this party can thus be fully trusted as it does not see any protocol messages.

We will assume that the dealer sends the parties secret shares of a *random* product:

$$\underbrace{[a], [b], [c]}_{\text{each party gets one share of } a, b \text{ and } c} \qquad \text{where} \quad a, b \xleftarrow{\$} \mathbb{F} \quad \text{and} \quad c = a \cdot b\,.$$

Then, a neat trick—first introduced by Beaver [Bea92]—allows us to compute $[x \cdot y]$ by using these shares of a random product (also sometimes called a *Beaver triple*).

The idea follows from the following somewhat counterintuitive rewriting of the product xy:

$$\begin{aligned} xy &= ((x-a)+a)((y-b)+b) \\ &= (x-a)(y-b) + a(y-b) + b(x-a) + ab \end{aligned}$$

The parties already have shares of $[a]$ and $[b]$, so if they had $x-a$ and $y-b$ *in the clear*, they could compute xy as follows:

$$[xy] = (x-a)(y-b) + [a] \cdot (y-b) + [b] \cdot (x-a) + [ab] \tag{1}$$

This is simply multiplication of shares by constants, addition of shares, and addition of constants, all of which we know how to do. So let's see how to put everything together.

Step 1: revealing blinded values. First, the parties will "blind" their inputs x and y with the random values a and b respectively. This is done by computing shares of $[\delta] = [x - a]$ and $[\varepsilon] = [y - b]$. This is just addition of secret-shared values, which we know how to do!

Then, the parties will reveal their shares to each other, so that each party learns $\delta = x - a$ and $\varepsilon = y - b$. These are one-time pad encryptions of x and y under the keys a and b respectively.

Step 2: computing the product of the blinded values. These blinded values can be multiplied in the clear since they have been revealed. So each party now holds the value:

$$\delta \cdot \varepsilon = (x - a) \cdot (y - b) \ .$$

Step 3: combining shares. Now we just have to do the steps outlined in (1).

1. Each party locally computes shares of $[a(y - b)]$ and $[b(x - a)]$, by multiplying their shares of $[a]$ and $[b]$ by the revealed values ε and δ respectively.
2. Each party locally computes shares of $[a(y - b) + b(x - a) + ab]$, by adding their shares of $[a(y - b)]$, $[b(x - a)]$, and $[ab]$ together.
3. Each party locally computes shares of $[xy]$ by adding the constant term $[\delta \cdot \varepsilon]$. (only one party needs to do this, as shown in Section 2.4).

It is not hard to show that this protocol is information-theoretically secure, if the random triplet $[a]$, $[b]$, $[c]$ is used only once (this is the same security argument as for one-time pad encryption).

2.5.2 Getting Rid of the Trusted Dealer

So we have seen that if the parties can get a random "multiplication triple" $[a]$, $[b]$, $[c]$ then they can compute $[xy]$ very efficiently. For now, we just assumed a trusted party for this functionality. But what if we don't want a trusted party? This is precisely what we wanted MPC for in the first place! But now the functionality we want to compute securely is very simple and specific: we just want to compute a secret-shared random product. There are several specialized protocols that can be used to do this.

Since generating these triples is expensive, it is common to split MPC protocols into a *pre-processing phase* and an *online phase*. The pre-processing phase is expensive, but it is independent of the parties' inputs and of the function f, since it only involves generating random triples. In the online phase, the parties use one of these triples for every multiplication gate in the circuit.

How can we generate these triples? We need public-key cryptography to do this. The most popular approach in modern MPC protocols (see e.g., SPDZ [DPSZ12]) is to use a (public-key) somewhat homomorphic encryption scheme (see the first lecture).

The rough idea is that the parties generate a public key, with a corresponding secret key that is secret-shared among the parties. Then the parties choose random shares $[a]$, $[b]$, encrypt them, and homomorphically evaluate the function $f(a_1, \dots, a_n, b_1, \dots, b_n) = (\sum_i a_i) \cdot (\sum_i b_i) = ab$ to obtain an encryption of ab. Finally, the parties run a distributed decryption protocol that results in each party obtaining a share of ab, without learning the secret key.

2.6 Analysis

Security. Note that the whole online phase of the protocol is information-theoretically secure: any collection of up to $n-1$ corrupt parties learns nothing about any secret-shared values, so we can trivially simulate their view by sampling uniformly random shares.

For the offline phase, semi-honest security follows from the security of the public-key primitives used to generate the Beaver triples, e.g., the semantic security of the somewhat homomorphic encryption scheme.

Efficiency. The amount of computation per party is linear in the number of gates in the circuit, i.e., $|C|$.

The amount of communication (both in the offline and online phases) is linear in the number of *multiplication* gates in the circuit. Thus, functions that require a lot of additions rather than multiplications will be a lot easier to compute in an MPC protocol.

3 Malicious Security

We have seen a very simple protocol for computing *any* function f without revealing anything else than the output to a (semi-honest) adversary who corrupts up to $n-1$ parties. Unfortunately, for practical applications we typically need malicious security, and that is where things get tricky and expensive.

There are numerous ways in which an adversary could deviate from the protocol specification:

- They could reveal incorrect output shares to bias the final result.
- More generally, they can use incorrect share values at any gate.
- They could drop specific messages.
- They could wait until all honest parties reveal their output shares, and then abort the protocol.
- etc.

The "ideal" execution of the protocol (with a trusted third party) actually captures many security properties beyond privacy, some of which can be tricky to formalize. These properties are trivially satisfied if the attacker follows the protocol specification, but can be violated by attackers who deviate from the protocol as above.

1. *Privacy:* the adversary learns nothing else than the output y.
2. *Correctness:* if an honest party outputs y, then $y = f(x_1, \ldots, x_n)$. In particular, all honest parties output the same value y.
3. *Input independence:* the adversary cannot choose their inputs as a function of the honest parties' inputs.
4. *Fairness:* if the adversary learns the output y, then all honest parties also learn y.

Properties (1)-(3) can be achieved by maliciously-secure MPC protocols, for any PPT adversary that corrupts up to $n-1$ parties. The first generation of MPC protocols achieved this by having all parties prove, in zero-knowledge, that they followed the protocol specification.

Modern MPC protocols use a much more efficient approach where secret shares are *authenticated* using a MAC scheme (see the homework!). Property (4) is much harder to achieve, and requires an honest majority of parties.

Efficiency. Generic MPC protocols are not used very often, as it is usually possible to design more efficient protocols tailored to a specific function f (e.g., the protocols we saw for commitments, ZK, etc.). There are also more efficient protocols for the special case of *two-party* computation, which we will not cover in this course.

Beyond confidentiality. The ideal notion of "privacy" that MPC aims at is that the adversary learns nothing about the honest parties' inputs *except for the output of the function*. As we will see in the second half of this course, there are many situations where this notion of privacy is actually too weak: the output of the function itself may reveal a lot of information about the honest parties' inputs. In such cases, we will need to depart from cryptography and rely on stronger statistical notions of privacy.

4 Beyond Additive Secret-Sharing

We have seen how n parties can share a secret among themselves so that any subset of $n-1$ corrupt parties learns nothing about the secret. This is the strongest form of privacy we can hope to achieve. But it also means that if a single party goes offline (or loses their share), the secret is lost.

We might thus want more general secret-sharing schemes that allow for recovery if at least t of the parties coordinate, for some $t \leq n$. This is called a *t-out-of-n* secret sharing scheme.

Definition 3 (t-out-of-n secret sharing scheme)**.** A secret sharing scheme over $\mathbb{Z}$ is a pair of efficient algorithms $(\texttt{Gen}, \texttt{Recons})$, such that

- $\texttt{Gen}(n, t, \alpha)$ returns shares $(s_1, s_2, \ldots, s_n)$ of the secret $\alpha \in \mathbb{Z}$, for $1 \leq t \leq n$.
- $\texttt{Recons}(s_{i_1}, \ldots, s_{i_t})$ takes in a subset of t shares and outputs the secret α.

The scheme should satisfy the following properties:

- *Correctness:* for every $\alpha \in \mathbb{Z}$, every set of n shares output by $\texttt{Gen}(n, t, \alpha)$, and every t-size subset $\{s_{i_1}, \ldots, s_{i_t}\}$ of the shares, we have that $\texttt{Recons}(s_{i_1}, \ldots, s_{i_t}) = \alpha$.
- *(Perfect) Privacy:* for every $\alpha, \alpha' \in \mathbb{Z}$ and every subset $S \subset [n]$ of size $t-1$, we have
$$\texttt{Gen}(n, t, \alpha)[S] \equiv \texttt{Gen}(n, t, \alpha')[S]$$
That is, the distribution of shares of α and α' are identical for any subset of $t-1$ parties. We can also define a *computational* security notion, where the distributions are computationally indistinguishable.

4.1 Shamir's Secret Sharing Scheme

Definition 4 (Shamir's Secret Sharing Scheme)**.** Let p be a prime, and $\alpha \in \mathbb{Z}_p$ be a secret to share.

- $\texttt{Gen}(n, t, \alpha)$: choose random coefficients $a_1, \ldots, a_{t-1} \xleftarrow{\$} \mathbb{Z}_p$ and define the polynomial:
$$f(X) = \alpha + a_1 X + \ldots + a_{t-1} X^{t-1} \in \mathbb{Z}_p[X] \,.$$
Note that f has degree at most $t-1$ and that $f(0) = \alpha$. For $i = 1, \ldots, n$, compute $y_i \leftarrow f(i) \in \mathbb{Z}_p$, and define $s_i = (i, y_i)$. Output the n shares $s_1, \ldots, s_n \in \mathbb{Z}_p^2$.
- $\texttt{Recons}(s_{i_1}, \ldots, s_{i_t})$: given the shares of a subset of t parties, interpolate the polynomial f and output $f(0) = \alpha$.

To prove correctness and privacy, we will use the following lemma:

Lemma 2 (Polynomial interpolation)**.** For every set of d points $(x_1, y_1), \ldots, (x_d, y_d) \in \mathbb{Z}_p^2$ such that $x_i \neq x_j$ for $i \neq j$, there exists a unique polynomial $f \in \mathbb{Z}_p[X]$ of degree at most $d-1$ such that $f(x_i) = y_i$ for every $i = 1, \ldots, d$.

Proof of Lemma 2. Let

$$f(X) = a_0 + a_1 X + \ldots + a_{d-1} X^{d-1} \quad \text{where} \quad a_0, \ldots, a_{d-1} \in \mathbb{Z}_p$$

be a polynomial of degree $d-1$. We require:

$$\begin{aligned} f(x_1) &= a_0 + a_1 x_1 + \ldots + a_{d-1} x_1^{d-1} = y_1 \\ f(x_2) &= a_0 + a_1 x_2 + \ldots + a_{d-1} x_2^{d-1} = y_2 \\ &\vdots \\ f(x_d) &= a_0 + a_1 x_d + \ldots + a_{d-1} x_d^{d-1} = y_d \end{aligned}$$

which we can write in matrix form as:

$$\underbrace{\begin{pmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^{d-1} \\ 1 & x_2 & x_2^2 & \ldots & x_2^{d-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_d & x_d^2 & \ldots & x_d^{d-1} \end{pmatrix}}_{V(x_1,\ldots,x_d)} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{d-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_d \end{pmatrix}$$

Notice that the matrix V does not depend on the y-values but only on the x-values. This $d \times d$ matrix is called the Vandermonde matrix $V(x_1, \ldots, x_d)$. Its determinant is

$$\det(V(x_1, \ldots, x_d)) = \prod_{1 \le i < j \le d} (x_j - x_i)\,,$$

which is non-zero iff $x_i \neq x_j$ for every $i \neq j$. This means that this system of linear equations has a unique solution $\vec{a} = V^{-1} y$, which gives us a unique polynomial f. In fact an explicit formula for the inverse of the Vandermonde matrix V^{-1} is known and can be used to compute the coefficient vector $\vec{a}$. □

Lemma 3. Shamir's secret sharing scheme is a t-out-of-n secret sharing scheme.

Proof.

- Correctness: by Lemma 2, a set of t shares uniquely determines a degree $t-1$ polynomial f. So the interpolated polynomial f has to be the one generated by `Gen` and so $f(0) = \alpha$.
- Privacy: Let $\texttt{View}_S(\alpha) = (y_1, \dots, y_{t-1})$ be the view of a subset $S \subset [n]$ of $t-1$ parties. We need to show that $\texttt{View}_S(\alpha) \equiv \texttt{View}_S(\alpha')$ for every $\alpha, \alpha' \in \mathbb{Z}_p$.

  Let $(z_1, \dots, z_{t-1})$ be arbitrary values in $\mathbb{Z}_p$. We have:

$$\begin{aligned}
\Pr_{a_1,\dots,a_{t-1}}[\texttt{View}_S(\alpha) = (z_1,\dots,z_{t-1})] &= \Pr[(\alpha, y_1, \dots, y_{t-1}) = (\alpha, z_1, \dots, z_{t-1})] \\
&= \Pr\left[V(0, x_1, \dots, x_{t-1}) \begin{pmatrix} \alpha \\ a_1 \\ \vdots \\ a_{t-1} \end{pmatrix} = \begin{pmatrix} \alpha \\ z_1 \\ \vdots \\ z_{t-1} \end{pmatrix} \right] \\
&= \Pr\left[\begin{pmatrix} \alpha \\ a_1 \\ \vdots \\ a_{t-1} \end{pmatrix} = \underbrace{V(0, x_1, \dots, x_{t-1})^{-1} \begin{pmatrix} \alpha \\ z_1 \\ \vdots \\ z_{t-1} \end{pmatrix}}_{\text{some fixed } u \in \mathbb{Z}_p^t} \right] \\
&= \frac{1}{p^{t-1}}
\end{aligned}$$

  So the distribution of $\texttt{View}_S(\alpha)$ is uniform over $\mathbb{Z}_p^{t-1}$ and independent of α, and thus identical to the distribution of $\texttt{View}_S(\alpha')$.

□

References

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology—CRYPTO'91: Proceedings 11*, pages 420–432. Springer, 1992.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play ANY mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218—229. Association for Computing Machinery (ACM), 1987.