Privacy Enhancing Technologies
Lecture 2 – Secure Multi-Party Computation

Florian Tramèr

AGENDA

1. Recap
2. Defining MPC
3. Computing on Secret Shared Data
4. Extensions

1 Recap on Commitments

In the previous lecture we saw an example of a very simple privacy functionality, where one party can commit to a secret and later reveal it. We also saw a first example of *computing* on secret data, with homomorphic commitments.

Today, we're going to focus on a much more general form of privacy protocol, where *multiple* (≥ 2) *parties* can compute *an arbitrary function* over their secret data.

2 Secure Multi-Party Computation

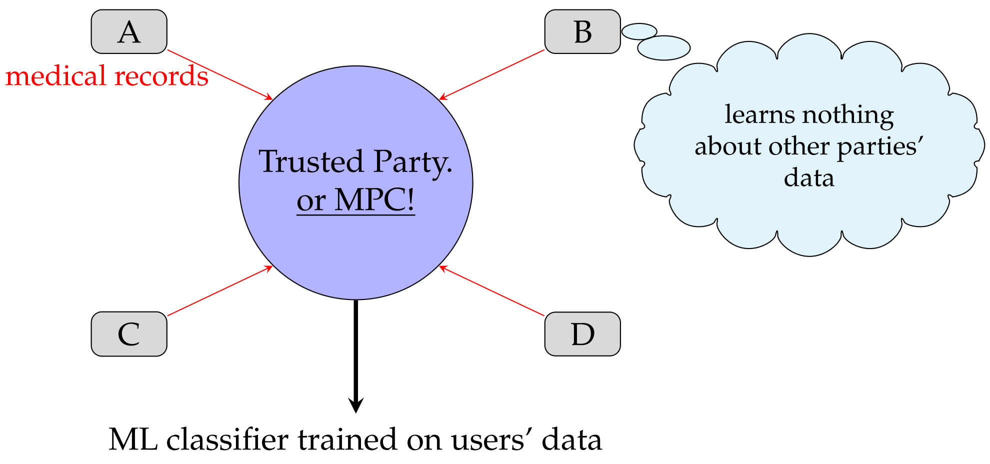
Today's lecture will be devoted to one of the most fascinating result of modern cryptography:

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

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

Example application: Private Machine Learning. Suppose that your 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 powerful keyboard prediction model on all of their text messages. In both of these cases, the parties want to obtain a good model by combining everyone's data, but they don't want to reveal their data to each other.

The "trivial" solution prescribed by cryptography is to have a trusted 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$\times$). And there are also specialized cryptographic protocols that are much more efficient for specific applications (e.g., secure messaging). But MPC has made enormous practical progress in the last few years, so it is likely that we'll see it used more and more in the future.

2.1 Defining MPC

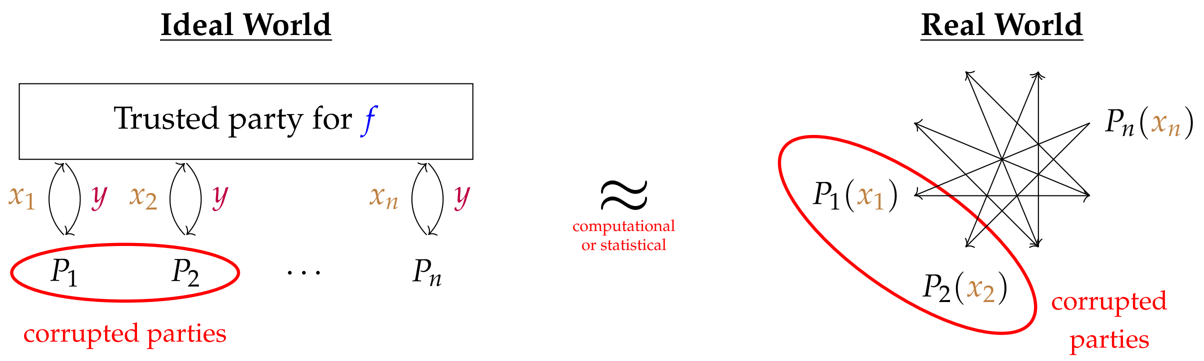
There are n parties $P_1, \ldots, P_n$ with inputs $x_1, \ldots, x_n$ that want to jointly compute a function

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

We can also generalize this to the case where the function generates multiple outputs $y_1, \ldots, y_n$ for 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.



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. And so our security goal will be that the adversary learns nothing more in the real world.

Let's formalize this security property. We'll start by introducing the notion of a *View* of an adversary, which is all the information that the adversary gets access to during the execution of the protocol (i.e., all the messages that the adversary receives and sends). To argue security, we then construct a *simulator*. This is a probabilistic algorithm that takes as input everything that the adversary sees in the ideal world, and outputs a simulated view that is indistinguishable from the real view (i.e., everything that the adversary sees during a real protocol execution).[1] What this means is that anything the adversary can learn in the real world could just as well have been simulated given just the ideal leakage.

Let's formalize this. Let C be the set of corrupt parties. We say a protocol securely computes f if there exists a simulator Sim such that for all inputs $x_1, \ldots, x_n$, we have

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

There are two main security models people consider for MPC:

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.
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

[1] A simulator is just an (efficient) algorithm that we define for our security proof. This is not an actual algorithm that we use in practice.

conceptually, and fairly efficient! We'll focus on these for now, and discuss some techniques to achieve malicious security at the end of the lecture.

3 Computing on Secret Shared Data

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.

So what's secret-sharing? It's just a simple way to "split" a secret into multiple chunks so that all chunks are needed to reconstruct the secret:[2]

Additive Secret Sharing

To share a secret $s \in \mathbb{F}_p$ (a field of prime order p) among n parties, sample random values $r_1, \dots, r_{n-1} \leftarrow \mathbb{F}_p$ and set $r_n = s - \sum_{i=1}^{n-1} r_i$. We use $[s]$ to denote the additive secret share of s:

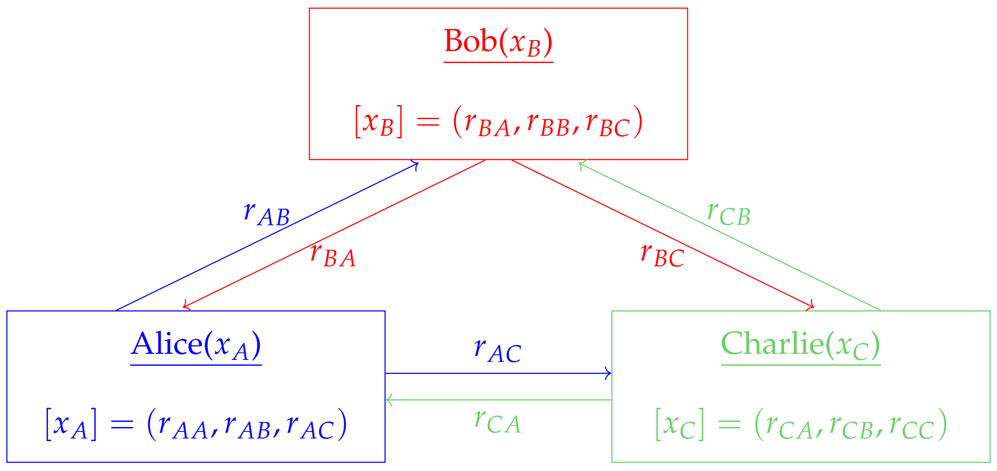
$$[s] = (r_1, \dots, r_n) \quad \text{such that} \quad s = \sum_{i=1}^{n} r_i\,.$$

Setup. 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 $\mathbb{F}_p$:

Definition 1 (Arithmetic Circuit)**.** An arithmetic circuit $\mathcal{C} : \mathbb{F}^m \to \mathbb{F}$ over a field $\mathbb{F}$ is a circuit consisting of binary gates applying the operations $\times, +$ over $\mathbb{F}$. (a standard Boolean circuit is a special case where $\mathbb{F} = \mathbb{Z}_2$, and the gates correspond to AND and XOR operations).

Any function f can be represented as an arithmetic circuit with moderate overhead.

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



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

[2]There are also more general *threshold* secret-sharing schemes where t-ouf-of-n shares are sufficient to reconstruct the secret, and any collection of $t-1$ shares reveals nothing [Sha79]. But we won't need those here.

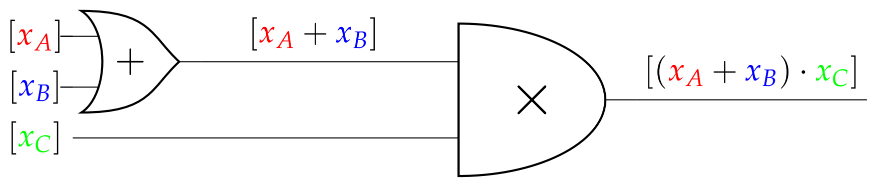
The protocol. Our MPC protocol is based on a simple observation:

> **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.

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.



Throughout the protocol, the values on the wires of the circuit are additively secret-shared:

Party A's shares	r_{AA}	r_{AB}	r_{AC}	r'_A	r''_A
Party B's shares	r_{BA}	r_{BB}	r_{BC}	r'_B	r''_B
Party C's shares	r_{CA}	r_{CB}	r_{CC}	r'_C	r''_C
Secret wire value	x_A	x_B	x_C	$x_A + x_B$	$(x_A + x_B) \cdot x_C$

Adding secret-shared numbers. 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:

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

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).

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

$$[kx] = [kx_1, kx_2, \ldots, kx_n] = k[x]\,.$$
$$[x+k] = [x_1 + k/n, x_2 + k/n, \ldots, x_n + k/n] = [x] + k/n\,.$$

Multiplying secret-shared numbers. The trick above does *not* work for multiplication gates.

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

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.

There are two main approaches to computing $[x \cdot y]$:

1. *The information-theoretic way:* If we use Shamir's secret sharing scheme [Sha79] rather than an additive secret sharing scheme, then we can actually compute $[x \cdot y]$ using a specialized protocol that provides information-theoretic, semi-honest security as long as less than $n/2$ of the parties are corrupt (it turns out that this is the best you can do without cryptographic assumptions).

   This gives us purely information-theoretic MPC, which is quite surprising and remarkable [BOGW88, CCD88]! If you're interested, you can see these lecture notes.

2. *The computational way:* What if we want to support $n/2$ or more corrupt parties? (e.g., what if $n = 2$? Or what if we use MPC for elections? Would you want to be part of the minority party?).

   Luckily, we can use public-key cryptography to obtain computationally secure MPC against an adversary who corrupts up to $n - 1$ parties [GMW87].

   This is the approach we'll take in this lecture. The main tools for multiplying shares are *oblivious transfer*, *garbled circuits* or *(somewhat) homomorphic encryption*. We won't cover how these work here, but we might talk about some of them in the homeworks.

Beaver's randomization trick. So multiplication gates are expensive: each one of them requires a sub-protocol that involves communication between the parties and expensive public-key cryptography.

A neat trick, first introduced by Beaver [Bea92], is to split the MPC into two phases: a *pre-processing phase* and an *online phase*. The pre-processing phase will be expensive, but it is independent of the parties' inputs and of the function f. In contrast, the online phase will be very fast, involving no public-key cryptography at all.

The trick consists in reducing the problem of multiplying specific secret-shared values $[x]$ and $[y]$ to the problem of multiplying *random* secret-shared values.

Suppose the parties already have 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 we can compute $[x \cdot y]$ as follows:

1. Each party locally computes shares of $[\delta] = [x - a]$ and $[\varepsilon] = [y - b]$. This is just addition, we know how to do this!
2. The parties 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. Since a and b are secret-shared themselves, no party learns anything about x or y.
3. Each party locally computes the following share:

   $$z_i = \frac{1}{n}\delta \cdot \varepsilon + b_i \cdot \delta + a_i \cdot \varepsilon + c_i.$$

   The output $[z] = (z_1, \ldots, z_n)$ is now secret-shared among the parties.

Why is this correct? Let's do the math:

$$\begin{aligned}
\sum_{i=1}^{n} z_i &= \left(\sum_{i=1}^{n} \frac{1}{n}\delta \cdot \varepsilon\right) + \left(\sum_{i=1}^{n} b_i\right) \cdot \delta + \left(\sum_{i=1}^{n} a_i\right) \cdot \varepsilon + \left(\sum_{i=1}^{n} c_i\right) \\
&= \delta \cdot \varepsilon + b \cdot \delta + a \cdot \varepsilon + c \\
&= (x-a)\cdot(y-b) + b \cdot (x-a) + a \cdot (y-b) + c \\
&= xy \ \cancel{-ay} \ \cancel{-bx} \ \cancel{+ab} + \cancel{bx} \ \cancel{-ab} + \cancel{ay} \ \cancel{-ab} + \cancel{ab} \\
&= xy
\end{aligned}$$

Essentially, what we're doing is: (1) blind x and y with the random values a and b so that we can reveal these values to all parties; (2) all parties compute $\delta \cdot \varepsilon = (x-a)\cdot(y-b)$ locally, which contains the product xy we care about; (3) cancel out a bunch of cross-terms by locally adding shares of a, b and c.

It isn't 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).

Generating Beaver triples. So we've seen that if the parties already have shares of a random "multiplication triple" $[a], [b], [c]$ then they can compute $[x \cdot y]$ very efficiently, without any public-key cryptography.

The bulk of the work is then to generate these triples in the first place. For this, we only need to know an upper bound on the number of multiplication gates in the function f we'll want to compute.

So in a pre-processing phase, the parties will use some heavy public-key machinery to generate many such random triples. Then, in the online phase they use one of these triples for every multiplication gate in the circuit.

4 Are We Done?

We've 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's where things get tricky and expensive.

Malicious security 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 $y = f(x_1, \dots, x_n)$. 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 malicious-security MPC protocols, for any adversary that corrupts up to $n-1$ parties. The first generation of MPC protocols achieved this by having all parties prove, in zero-knowledge (see next lecture), that they followed the protocol specification. Modern MPC protocols use a much more efficient approach where secret shares are *authenticated* using a MAC (message authentication code) 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. This is especially true for cryptographic functions based on number theoretic constructions (e.g., RSA, elliptic curve cryptography, etc.)

There are also more efficient protocols for the special case of *two-party* computation, which we won't 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.

References

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

[BOGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 1—-10. Association for Computing Machinery (ACM), 1988.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 11–19. Association for Computing Machinery (ACM), 1988.

[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.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.