# Lecture 15 : Secure Multiparty Computation (MPC)

5/18/21

# Plan

Multiparty computation (MPC): Intro & examples

Defining security for MPC

How to build MPC

Malicious security for MPC

Previously on CS355...

- 2 party protocols for proofs
- 2 party protocols for arbitrary functions
- Secret sharing
  - ↳ a "multiparty" protocol (for an arbitrary # of parties)

Q: Can we do more in the multiparty setting than just sharing secrets?

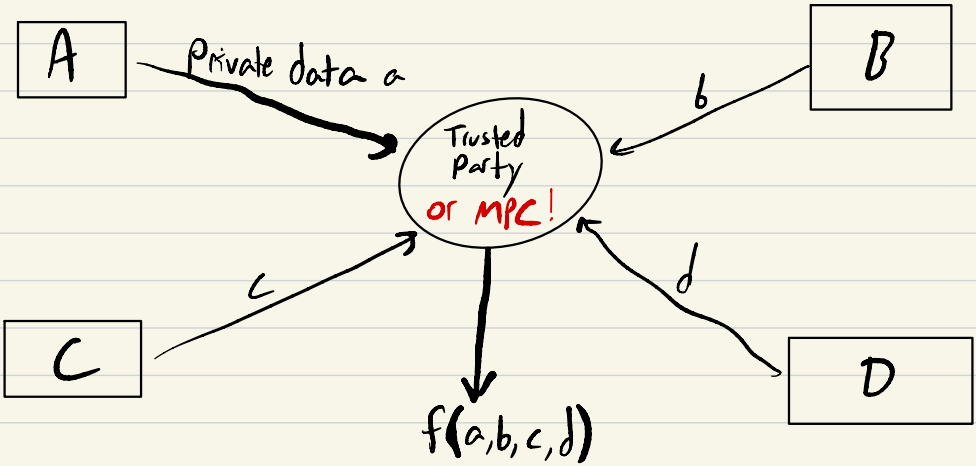
Can we compute on them?

↳ Note: threshold decryption is a special case of computing on secret-shared data

# Multiparty Computation (MPC)

"Any function that can be securely computed with a trusted third party (TTP), can also be securely computed without one."

Example: Private data analysis




Recent use-cases:

2017-present: Boston Women's Workforce Council

measures aggregate gender/racial wage gap in Boston area

why MPC? Businesses willing to help with data collection but don't want to be singled out.

2015: Estonia used MPC to merge tax, employment, and higher education data to understand why so many IT students drop out.

why MPC? Law prevents sharing individual data, analysis on aggregate data cannot account for smaller groups in data set.

Other examples of MPC:

- training/running ML models
- E-voting
- Private auctions
- Generating secret parameters for other protocols

  e.g. CRS for a ZK-SNARK
- Setting up a threshold encryption scheme without any party knowing the secret key!

  AKA "Distributed Key Generation"

⋮

Q: MPC is so general! Why not use it for <u>everything</u>?

A: In general, it's not very efficient! (eg. 100x–1,000x+ overhead)

- but keeps improving!
- More efficient special purpose protocols for some computations

# 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

$$y = f(x_1, \ldots, x_n)$$

Can generalize so each party gets its own $y_i$

<u>The adversary corrupts a subset of the parties</u>

and makes them collude to break security of the protocol

↳ In the 2-party setting, the adversary would always corrupt just 1 party
Q: Why can't the adversary corrupt all the parties?

Two main security models for MPC (Same as 2PC)

**Semi-honest:** The corrupted parties follow the protocol specification exactly. After the protocol completes, they look at their transcripts and try to extract information about the honest parties' inputs.

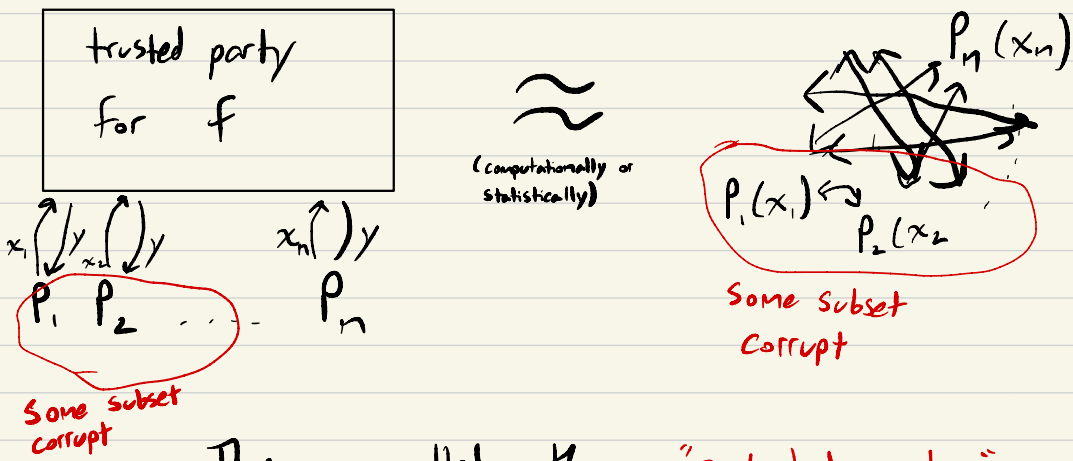
**Malicious:** The corrupted parties may arbitrarily deviate from the protocol specification at any time to learn extra information about the honest parties' inputs or fool them into producing the wrong output.

We'll mainly focus on the semi-honest setting.

# Formalizing Security for Semihonest MPC

Informally: "anything the adversary learns in an execution of the MPC protocol, it could also have learned if all parties were interacting with a trusted third party"

**ideal world** $\approx$ **real world**




This is called the "real-ideal paradigm"

What does the adversary learn in the real world?

- inputs of corrupted parties
- outputs of trusted party $(f)$ ← implicitly captures correctness too!

Formally, if $C$ is the set of corrupt parties, there exists an efficient simulator Sim such that

$$\mathrm{Sim}(C, \underbrace{\{x_i : i \in C\}}_{\text{inputs of corrupt parties}}, \underbrace{y = f(x_1, \ldots, x_n)}_{\text{output of computation}}) \approx_c \underbrace{\{\mathrm{View}_i : i \in C\}}_{\text{View of A in real execution of protocol}}$$

# How to build MPC: Computing on Secret-shared data

Recap: additive secret sharing

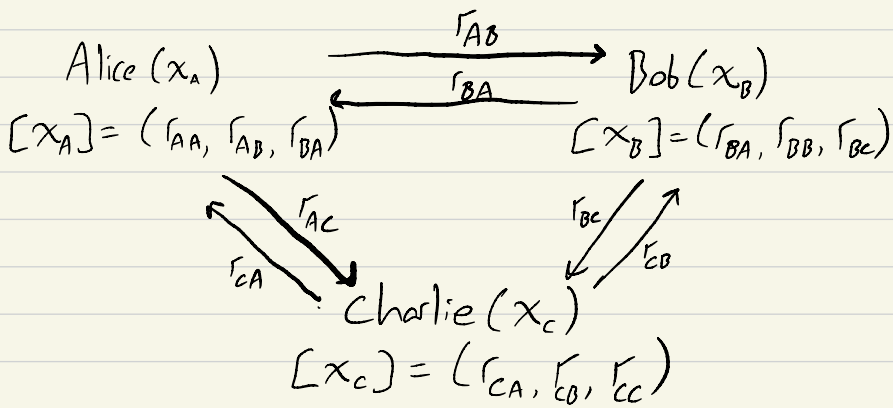
to share $s \in \mathbb{F}_p$ among $n$ parties, sample $r_1, \ldots, r_{n-1} \xleftarrow{R} \mathbb{F}_p$ and set $r_n \leftarrow s - \sum_{i=1}^{n-1} r_i$

We use $[s]$ to denote additive secret shares of $s$:

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

## MPC Setup:

- Each party has input $x_i \in \mathbb{F}_p$
- Function $f$ represented as arithmetic circuit over $\mathbb{F}_p$ (i.e., $+, *$ gates over $\mathbb{F}_p$)
- Parties start by secret sharing their inputs




Observation: to get a semihonest MPC protocol, all we need is the ability to add and multiply secret shared data!

High-level MPC protocol:

1. each party secret shares its input with every other party
2. for each add/multiply gate on inputs $[x]$, $[y]$, parties compute $[x+y]$, $[xy]$ respectively
3. At end of circuit parties publish their shares of output so everyone can reconstruct output

<u>Adding secret-shared data is easy!</u>

Component-wise addition

Given shares of $x$ and $y$, $[x+y] = [x] \overset{+}{+} [y]$

why? if $[x] = (x_1, \dots, x_n)$ where $\sum_{i=1}^{n} x_i = x \in \mathbb{F}_p$
$[y] = (y_1, \dots, y_n)$ where $\sum_{i=1}^{n} y_i = y \in \mathbb{F}_p$

Then $[x+y] = (x_1+y_1, \dots, x_n+y_n)$ satisfies $\sum_{i=1}^{n} (x_i+y_i) = x+y \in \mathbb{F}_p$

Similarly:
- multiplication by a non-secret value: $k \cdot [x] = (k \cdot x_1, \dots, k \cdot x_n) = [kx]$
- addition with a constant: $k + [x] = (x_1 + \frac{k}{n}, \dots, x_n + \frac{k}{n}) = [k+x]$
  could also do $(x_1 + k, x_2, \dots, x_n)$

Security is information-theoretic! Any $n-1$ parties learn nothing about $x$ & $y$

How do we multiply secret shared data?

Not as easy!! $[x \cdot y] \neq [x] \cdot [y] = (x_1 y_1, \dots, x_n y_n)$

i.e. $xy \neq \sum_{i=1}^{n} x_i y_i$

→ the parties will have to interact!

Approach 1:

- purely information-theoretic! ← WOW!
- assumes adversary corrupts $< \frac{n}{2}$ parties ← "honest majority" setting
- extremely efficient
- see CS355 2019 notes if interested

What if Adv can corrupt $\geq \frac{n}{2}$ parties?

→ e.g. 2 party case?

→ would you vote for a minority party if coalition of $\geq 50\%$ of voters could reveal who you voted for?

Approach 2:

- computationally secure
- adversary can corrupt up to $n-1$ parties
- uses <u>Oblivious transfer</u> or <u>Somewhat homomorphic encryption</u>

↑ encryption scheme that lets you homomorphically evaluate polynomials of bounded degree

(coming soon)

# Multiplying shares with Beaver Triples [Beaver '91]

Suppose the parties have shares of a random product:

$$\underbrace{[a], [b], [c]}_{\text{each party has one share of each.}} \quad \text{where } a, b \xleftarrow{R} \mathbb{F}_p \text{ and } c \leftarrow ab$$

(We'll discuss how to get these shares later)

To multiply $[x]$ and $[y]$:

1. each party locally computes $[x-a]$ and $[y-b]$
2. parties jointly reconstruct $\underbrace{\varepsilon = x-a}$ and $\underbrace{\delta = y-b}$
3. each party locally computes

$$z_i \leftarrow c_i + x_i \cdot \delta + y_i \cdot \varepsilon - \frac{\varepsilon\delta}{n}$$

$$[z] = (z_1, \ldots, z_n)$$

one-time pad encryptions of $x, y$, so privacy is preserved!

information-theoretic security! yay!

<u>Correctness:</u> $z = \sum_{i=1}^{n} z_i = \sum_{i=1}^{n} c_i + x_i\delta + y_i\varepsilon - \frac{\varepsilon\delta}{n}$

$$= c + x\delta + y\varepsilon - \varepsilon\delta$$
$$= c + x(y-b) + y(x-a) - (y-b)(x-a)$$
$$= \cancel{ab} + xy - \cancel{xb} + \cancel{yx} - \cancel{ya} - \cancel{xy} + \cancel{ya} + \cancel{bx} - \cancel{ab}$$
$$= xy$$

# How to generate Beaver triples?

→ Oblivious Transfer
→ Garbled Circuits } See HW!
→ Somewhat homomorphic encryption

Note: Generating beaver triples is expensive! (public-key crypto)

How many triples do we need? One per multiplication gate!
↳ what happens if we reuse a triple? two-time pad!

Optimization: preprocessing

Offline phase: parties generate $M$ random triples
↑ expensive but independent of user inputs & function

Online phase: ← no expensive crypto, but lots of communication

- parties secret share inputs
- parties securely compute $f$ using pre-generated triples ← circuit for $f$ has $\leq M$ mults
- parties reconstruct output

Performance considerations:

- computation & communication depends mainly on # of multiplications
- # of communication rounds depends on multiplicative depth of circuit

# Malicious Security

Some ways adversary could deviate from protocol:

- output incorrect shares in final step
- add/multiply shares incorrectly
- change input to protocol
- drop messages
- refuse to output their share
- ...

Many potential security goals: (Some tricky to formalize)

1) privacy: parties only learn the output of $f$
2) Correctness: if a party receives output $y$, then $y = f(x_1, \dots, x_n)$
3) input independence: corrupt parties' inputs can't depend on honest parties' inputs
4) fairness: if adversary learns output, so do honest parties

Some things we know (stated here without proof):

(4) fairness only possible if there is an honest majority → fair 2PC impossible :(

(1-3): possible w/ information-theoretic security if $\geq \frac{2}{3}n$ parties honest

possible w/ computational security if at least 1 party is honest!

- "classical" approach [Goldwasser, Micali, Wigderson, 1987]
  1) parties commit to inputs
  2) in each step, parties prove in zero-knowledge that they followed the protocol
- "modern" approach ["SPDZ" protocol: Damgard, Pastro, Smart, Zakarias, 2011]
  ↳ compute on shares of authenticated data