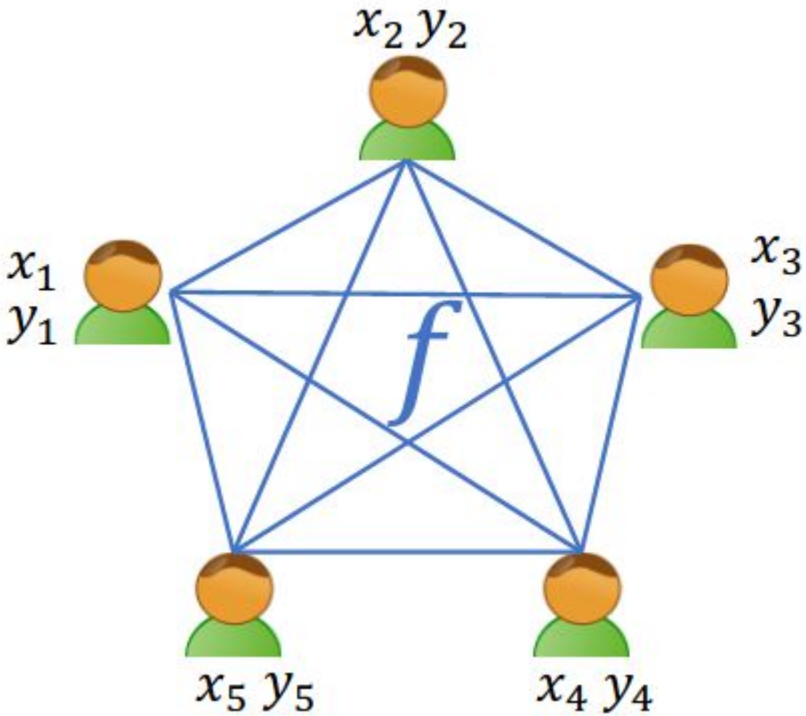
# CS-523 Project 1

Secure Multi-Party Computation

# Secure Multi-Party Computation (Week 2 Lecture)



# Project in a Nutshell

High-level parts:

1. Implementation of an SMC protocol
2. Evaluation of its performance
3. Application of the protocol

Deliverables: 2-page report and code

Submission deadline: 2 April 2021

# Implementation Goal*: Convenient Python Library for SMC

*very simplified

```python
# Define secrets
alice_secret = Secret()
bob_secret = Secret()
# Define arithmetic circuit (=“expression”)
expr = alice_secret + bob_secret

# Alice runs protocol, communicating with Bob and third parties
run_protocol(expr, value_dict={alice_secret: 5})
# Bob runs protocol, communicating with Alice and third parties
run_protocol(expr, value_dict={bob_secret: 12})
```

# SMC with Secret Sharing

## $N$-Players SMC — Additive Secret Sharing

- A technique to protect secrets which allows for computations to be carried out

For a secret $s \in \mathbb{Z}_q$, we can compute its $N$ additive secret shares $s_0, s_1, \dots, s_{N-1}$ :

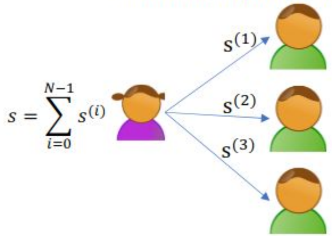
- Sample $s^{(0)}, s^{(2)}, \dots, s^{(N-1)} \in \mathbb{Z}_q$ uniformly at random
- Set $s^{(0)} = s - \sum_{i=0}^{N-1} s^{(i)} \ mod\ q$
- We denote by $[s] = \{s^{(0)}, \dots, s^{(N-1)}\}$ the sharing

Reconstructing the secret is done by summing all shares together *mod q*

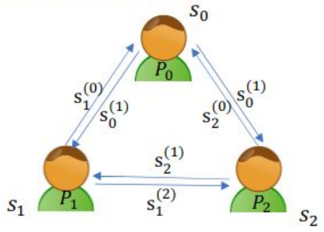
When providing each player with a share of the secret, $s$ benefits from strongest link security:

→ An adversary must corrupt all players in order to reconstruct the secret

→ In a multiparty computation context, private **inputs can be shared** in the same way




"Secret outsourcing"




"Input sharing "

36

# Implementation: Secret Sharing and Addition

## Additive Secret Sharing – Operations

- Arithmetic operations can be carried out on additive secret shared data
- $+$ operation require no message exchange

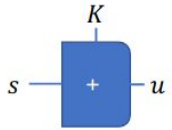
  Given $s = \sum_{i=0}^{N-1} s^{(i)}$ and $t = \sum_{i=0}^{N-1} t^{(i)}$ two secret shared variables, party $i$ holds share $s^{(i)}$ and $t^{(i)}$

<u>ADD Protocol</u>:

Each party $i$ :

1. Computes $u^{(i)} = s^{(i)} + t^{(i)}$


→ After Protocol ADD, the parties **collectively know** the secret shared variable $u = \sum_{i=0}^{N-1} u^{(i)}$




<u>ADD-K Protocol:</u>

In order to add a constant $K$, each party $i$ computes $u^{(i)} = \begin{cases} s^{(i)} + K \; if \; i = 0 \\ s^{(i)} \; otherwise \end{cases}$

→ After Protocol ADD, the parties **collectively know** the secret shared variable

$$u = \sum_{i=0}^{N-1} u^{(i)} = s + K$$

37

# Implementation: Multiplication using Beaver Triplet Scheme

## Additive Secret Sharing – Operations

- $\times$ operation is more complex: requires a secure multiparty computation protocol

  Used techniques and efficiency depends on the number $N$ of parties and the adversarial model

  An example of the multiplication using **Beaver triplets**.

  Consider that the parties have an additive secret sharing of three **uniformly random** values $a, b$, and $c$ such that

  $$a.b = c \in \mathbb{Z}_q$$

  - Given two shared inputs $[x]$ and $[y]$, proceed to:
    1. Each party $i$ computes $[x - a]$, and broadcast their share i.e. $x^{(i)} - a^{(i)}$
    2. Each party $i$ computes $[y - b]$, and broadcast their share i.e. $y^{(i)} - b^{(i)}$
    3. Each party $i$ reconstructs $(x - a)$ and $(y - b)$ and computes **locally :**

    $$[z] = [c] + [x].(y - b) + [y].(x - a) - (x - a)(y - b)$$


38

# What we provide

1. Skeleton of the implementation in Python 3
2. Test suite that your implementation has to satisfy (`test_integration.py`)
3. Code that handles networking and communication

https://github.com/spring-epfl/CS-523-public

# Tour of the skeleton

Your implementation should normally reside in these files:

- **`expression.py`**—Tools for defining arithmetic circuits (=“expressions”)
- **`secret_sharing.py`**—Secret sharing scheme
- **`ttp.py`**—Trusted parameter generator for the Beaver multiplication scheme
- **`smc_party.py`**—SMC party implementation

Some code that will help you out:

- **`protocol.py`**—Specification of SMC protocol
- **`communication.py`**—SMC party-side of communication
- **`server.py`**—Trusted server to exchange information between SMC parties

Tests:

- **`test_integration.py`**—Integration test suite. Your implementation *must pass these.*
- ...Some templates of test files for you to start from

# Communication

Methods:

- **`send_private_message`**`(receiver, label, message)`
- **`retrieve_private_message`**`(label)`

- **`publish_message`**`(label, message)`
- **`retrieve_public_message`**`(sender_identifier, label)`

- **`retrieve_beaver_triplet_shares`**`(operation_identifier)`

# Evaluating your implementation

Measure costs = runtime and bytes communicated

- Effect of the number of parties on costs
- Effect of the number of additions on costs
- Effect of the number of multiplications on costs

# Application

Requirements:

- Involves multiple parties
- Uses all kinds of operations

Implementation:

- Test the correctness of your implementation of the circuit

Analysis:

- Motivation for this application
- Threat model
- Privacy properties
    - SMC guarantees that parties learn nothing but the output.
      But the output itself can also leak private information! Cf. Lecture on differential privacy soon

# Project in a Nutshell

Good luck and have fun!

High-level parts:

1. Implementation of an SMC protocol
2. Evaluation of its performance
3. Application of the protocol

Deliverables: 2-page report and code

Submission deadline: 2 April 2021