Report on Private Multi Party Computation using Yao’s protocol

Achraf Guenounou

achrafgu@edu.aau.at

June 2024

Introduction

This report focuses on detailing the technical functionalities of implementing Yao's protocol for private Multi-Party Computation (MPC). MPC allows the computation of functions using datasets owned by different parties while ensuring the privacy of the data.

This particular implementation builds on the garbled-circuit library developed by Olivier Roques and Emmanuelle Risson, available on GitHub. However, it extends it to address the specific problem outlined below.

Problem description

This implementation of Yao's protocol is tailored to compute the common value in two sets, each consisting of four integers, each ranging from 0 to 15, divided between two parties: "Alice" and "Bob." Alice functions as the "garbler," responsible for generating the circuit and creating the garbled gates to be transmitted. Bob serves as the "evaluator," tasked with determining the correct value from the garbled tables and communicating it back to Alice. The protocol ensures that, upon completion, both parties will know the common value, while all others values remain confidential.

Computation and communication description

The computation follows the classic idea of the MPC protocol. For ease of implementation and according to one of the guidelines provided by the supervising professor of this project, it is considered that Alice and Bob can each choose only a fixed number of four elements that vary in the range 0-15, meaning they have a binary encoding of at most 4 bits.

Alice’s communication

Alice is responsible for generating the circuit, which involves creating the garbled tables that Bob will later evaluate. She must also send the circuit to Bob in advance. Given that the circuit is fixed, there is no need for initial communication with Bob to generate the garbled tables, so it’s given directly to Alice via console.

Alice's workflow is outlined in the pseudocode below.

Alice communication pseudocode

```
alice_input ← read_input(alice)
garbled circuit ← garble(circuit)
send(bob, garbled circuit)
bob_keys ← generate_keys()
alice_binary_input ← convert_to_binary(alice_input)
alice_encrypted_input ← encrypt(alice_binary_input)
send(ot, alice_encrypted_input, bob_keys)
result ← receive(ot)
verify_result(result)
```

Note: in send and `receive` operations the actor that interacts with Alice is underlined

Bob’s communication

Bob is tasked with evaluating the circuit result using the garbled tables and the secret keys corresponding to his inputs, which were generated by Alice. These keys are acquired through the Oblivious Transfer (OT) protocol.

Bob's workflow is outlined in the pseudocode below.

Alice communication pseudocode

```
bob_input ← read_input(bob)
garbled_circuit ← receive(alice)
bob_binary_input ← convert_to_binary(bob_input)
send(ot, bob_binary_input)
bob_keys ← receive(ot)
result ← evaluate(garbled_circuit, bob_keys)
send(ot, result)
verify_result(result)
```

Note: in `send` and `receive` operations the actor that interacts with Bob is underlined

Circuit functioning and structure

The circuit was designed to perform a bitwise comparison between every number in the first set and every number in the second set. The circuit logic was initially developed on Logisim and subsequently transcribed into a JSON dictionary following the guidelines of the reference repository.

For the comparison between two numbers, a module was created (Figure 1) capable of taking two 4-bit values as input, returning two pieces of information: a 4-bit number, which represents the common value, and a match bit, which indicates whether the comparison was successful or not. This bit is useful because, both in the case of no match and if the common number is zero, the circuit would return “0000” as the output number. Thanks to this bit we can determine if there has been a match.

This module was then used as a basis to implement the remaining comparisons (Figure 2). There was still one problem to solve: which element to return in the set if there were multiple common elements, without revealing additional information. It was decided to return the first common element found, using the match bit signal to set to zero the subsequent values in case the match was positive.

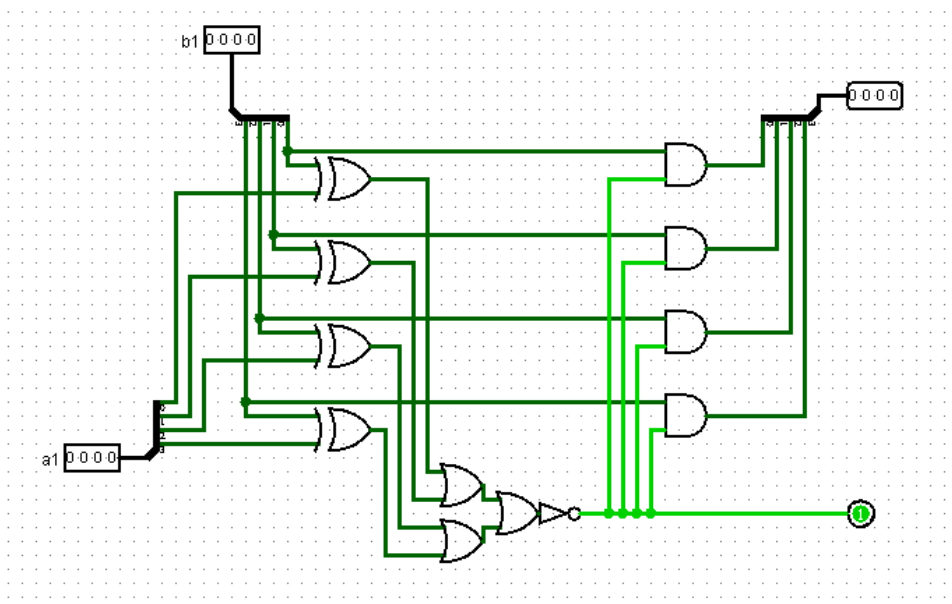

Figure 1: module used for comparing two 4-bit values

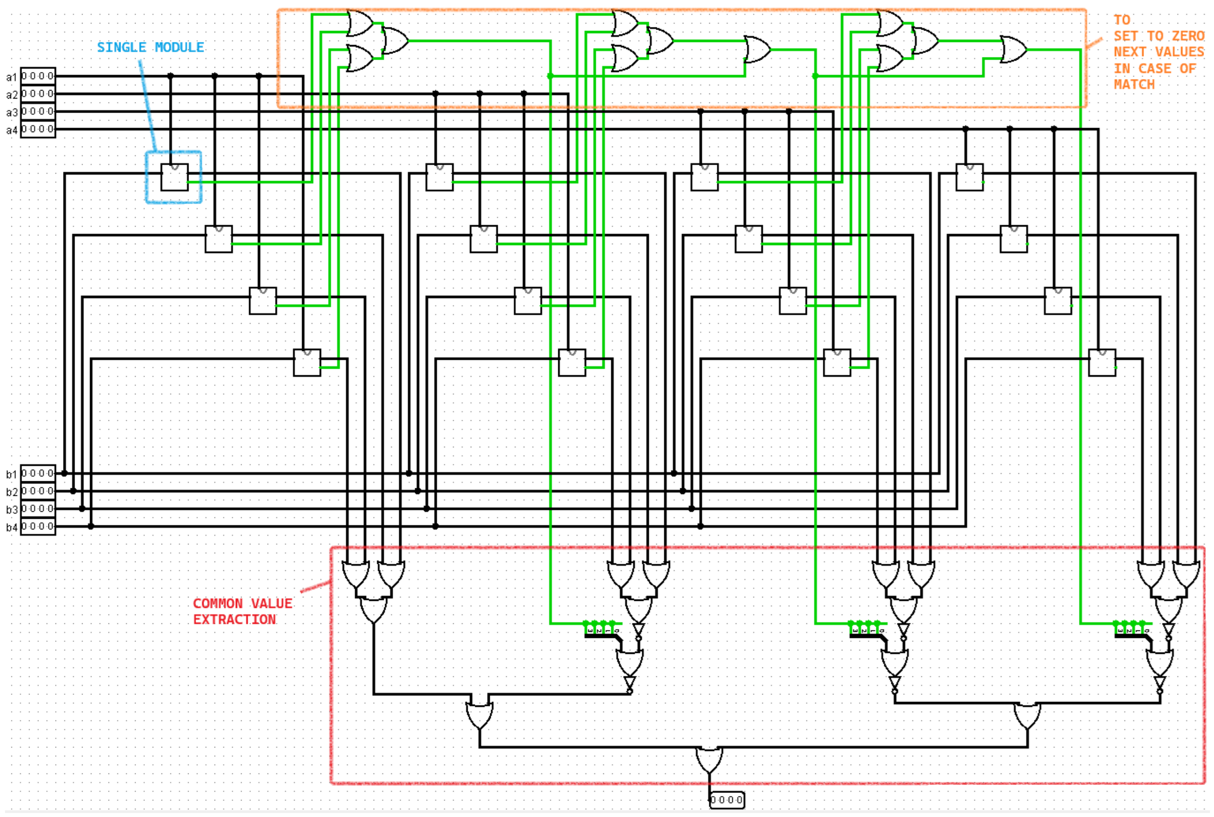

Figure 2: full circuit for comparing two sets of 4 elements each of 4 bits

Functionality and Project Structure

The project is organized into various files, each containing specialized classes and functions dedicated to specific roles within the protocol.

`alice.txt` and `bob.txt`

These two files are used to store the inputs of alice and bob before starting the computation. For correct reading, the input should be formatted as follows: 4 values for each, in the range 0-15, written into the respective files, all separated by a space and on the same line.

`requirements.py`

This file contains all the functions required for the implementation of the project. The functions are described below:

- **`print_alice_to_bob`**. This function prints all the data that Alice sends to Bob into a file located at ‘`intermediate-outputs/output_alice_to_bob.txt`’. It also generates a JSON file containing the garbled tables at ‘`intermediate-outputs/alice_circuit_produced.json`’.
- **`alice_bob_OT`**. This function saves messages related to the Oblivious Transfer between Alice and Bob into the respective file. Two different files are generated, “`alice_ot.txt`” and “`bob_ot.txt`”, both inside the folder “`intermediate-outputs/ot-communications`”.
- **`bob_alice_mpc_computed`**. According to the signature proposed by the project's `format.py` file, the function was supposed to compute the output for the common element. However, since this functionality is already provided by the `yao.evaluate()` function from the library, `bob_alice_mpc_computed` merely retrieves the result when the respective parties receive it and prints it. Initially, two separate functions were designed for Alice and Bob, but as they served the same purpose, a single function was created. This function is called immediately after the evaluation for Bob and right after receiving the result for Alice.
- **`verfiy_output`**. This function verifies the correctness of Yao’s protocol computation by comparing its result to the common element determined by reading the contents of the input files where Alice and Bob store their elements and selecting the first element in Alice's set that is also present in Bob's set. It returns 1 if it is the same else 0.

`main.py`

The main file for the execution of the two parties. It contains both the Alice and Bob classes. It also includes the "YaoGarbler" class, which serves as a base class from which both Alice and Bob inherit.

`ot.py`

This file includes all the necessary functions and classes for performing the Oblivious Transfer (OT). It is largely the same as the version available on the GitHub repository, with a minor modification: the ‘`send_result`’ function now also returns the result to Bob, allowing him to know the output and print it.

`util.py`

This file contains utility functions and classes aimed at enhancing clarity and simplifying the functions of other classes. It mirrors the version found in the GitHub repository.

`yao.py`

This file implements the evaluation phase of Yao’s protocol and it is identical to the version found in the GitHub repository.

Script usage

Environment specifications

The protocol was implemented using Python version 3.10.1 therefore, functionality with other versions of Python is not guaranteed. The following additional libraries are required for this project in order to work correctly:

- cryptography 42.0.5
- pyzmq 25.1.1
- sympy 1.12

These packages are essential for running the standard version of the GitHub library. The number next to each library indicates the version it was tested with, and it is highly recommended to use that version (or a later one). All of these packages can be installed via pip3:

```
pip3 install cryptography pyzmq sympy
```

Other packages that are used (such as “sys” and “hashlib”) are part of the standard Python library and are already available with each Python3 installation.

Running the script

To run the script, the user must open two separate terminals in the project root directory. These terminals will act as the two different parties involved in the protocol.

Before starting Alice and Bob it is necessary to insert 4 values for each in the range 0-15 into the respective files, all separated by a space and on the same line.

Then it is sufficient to execute

```
python main.py alice -c circuits/comp-eq.json
```

in order to run Alice (indicating after `-c` the path to the JSON file representing the circuit) or

```
python main.py bob
```

to run Bob.

If the inputs are not in a valid format the process will abort asking to correct the file, otherwise the MPC communication between the two will begin which will end by returning the result of the comparison on both terminals.

Running example (Alice)

```
…\Achraf_Guenounou_Project\garbled-circuit\src> python
main.py alice -c circuits/comp-eq.json

============ READING ALICE INPUT ============

Input read from Alice's file : [1, 6, 0, 0]

============ STARTING YAO PROTOCOL ============

Initialized Alice's OT log file at the path
intermediate-outputs/ot-communications/alice_ot.txt

=== WAITING FOR RESULT COMPUTATION OF common_element_4x4 ===

Result computed according to Yao’s protocol:
        - match bit value is 1
        - common number bits are [0, 1, 1, 0]

============ VERIFYING OUTPUT ============

The comparison between sets was successful. The common
element is 6.

============ END OF COMMUNICATION ============
```

Running example (Bob)

```
…\Achraf_Guenounou_Project\garbled-circuit\src> python
main.py bob

============ READING BOB INPUT ============

Input read from Bob's file : [9, 0, 6, 7]

============ STARTING LISTENING FOR MESSAGES ============

Initialized Bob's OT log file at the path
intermediate-outputs/ot-communications/bob_ot.txt

============ INITIALIZING CIRCUIT COMPUTATION ============

Received common_element_4x4
Result computed according to Yao’s protocol:
        - match bit value is 1
        - common number bits are [0, 1, 1, 0]

============ VERIFYING OUTPUT ============

The comparison between sets was successful. The common
element is 6.

============ END OF COMMUNICATION ============
```