Project documentation

Lorenzo Zanolin

May 23, 2023

1 Introduction

The aim of this project is the study of Yao's protocol [3] and an useful application of it. More precisely, we will implement Secure multi-party computation; this field has the goal of creating methods for parties to jointly compute a function over their inputs while keeping those inputs private [2]. In this project, the function we decided to implement is the *8 bit sum*.

2 Analysis

We can focus on the sum computed by two parties, say Alice and Bob. The goal is to use *MPC* to calulate the result while avoiding each member to know the others values; to do that we need an external component, called *OT*.
This protocol allows two parties, Alice who knows x and Bob who knows y, to compute jointly the value of $f(x, y)$ in a way that does not reveal to each side more information than can be deduced from $f(x, y)$ [1]. As already written, f is the 8-bit sum.
Both Alice and Bob have a set of integers and need to compute the sum of everything. Two new roles are introduced: Alice is the *garbler* and has the responsibility of creating the garbled circuit, while Bob is the *evaluator*. They will behave as follows:

1. Alice creates the garbled circuit and the tables and send them to Bob.
2. Alice select her set, via input.
3. Bob select his set, via input.
4. Both interact with each other using OT.
5. Bob evaluate the function result and sends it to Alice.
6. Alice checks the correctness of the result, and sends it to Bob (in clear).

3 Implementation

We will present how to implement the *circuit* and how to make Alice and Bob comunicate securely.

3.1 Circuit

We will present briefly the 8-bit sum circuit. There are three basic components in this construction:

- *Half Adder*: used to sum the right-most digit;
- *Full adder*: used to sum a generic digit in the number, ranging from position 1 to 8. It receives in input also carry of the previous sum.
- *If then else*: used to prevent overflow, induced by the 2-complement sum.

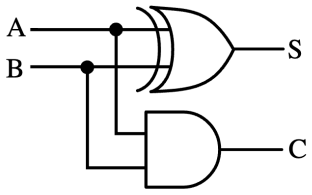

Figure 1: Half Adder

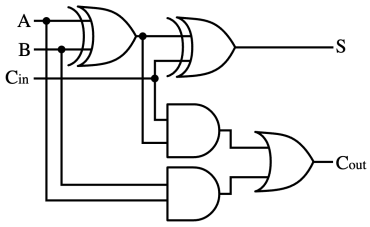

Figure 2: Full Adder

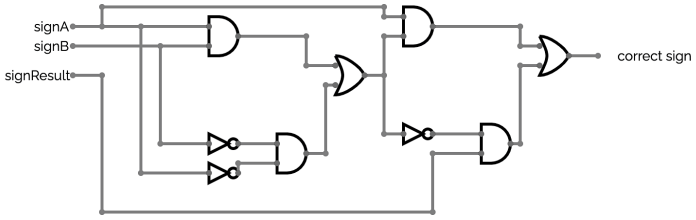

Figure 3: If then else

[1]

We then proceede creating the circuit by wiring 7 full adders, an half adder and the if-then-else together, as represented in Figure 4.

Since we are using 8 bit to represent the sum of each set, it immediatly follows that the sum of each set must be smaller than 256. In this case we are dealing with *integers*,

[1] 1 was taken over `https://upload.wikimedia.org/wikipedia/commons/1/14/Half-adder.svg` 2 was taken over `https://upload.wikimedia.org/wikipedia/commons/a/a9/Full-adder.svg`.

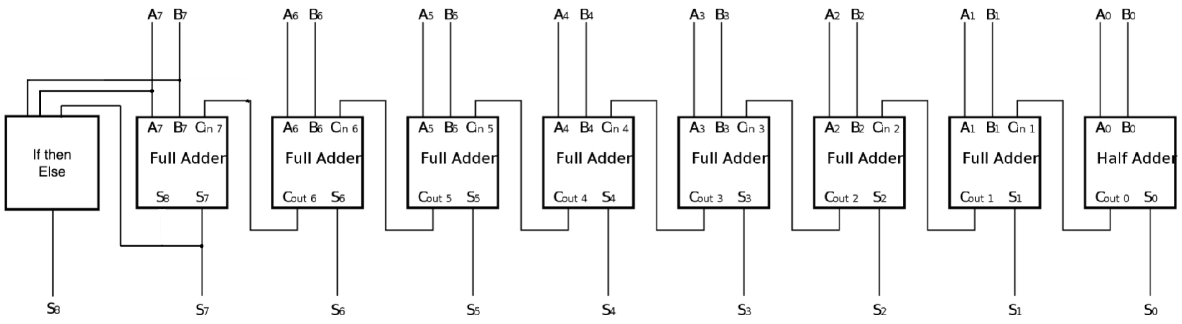

Figure 4: 8-bit Adder

thus we also need to consider negative numbers; to do that we are using *2-complement*. In this case we are considering numbers that belong to [-128,127]. The final sum has 9 bit representation, thus the result that we can obtain belongs to [-256,254]. The *if then else* component is used to prevent overflow situations induced by the sum, i.e.when you add two numbers with the same sign.

3.1.1 Project structure

The project will be developed using *Python 3.9.10* and we will use functions provided in the GitHub repo `https://github.com/ojroques/garbled-circuit`.
There is also a dedicated GitHub page related to this project at `https://github.com/lorenzozanolin/garbledCircuit`.

The project has the following structure:

```
src/.
├── Makefile
├── images                    This directory contains the images used.
│   └── ...
├── circuits                  This directory contains the circuit used.
│   └── add.json
├── code                      This directory contains the code used.
│   ├── util.py
│   ├── yao.py
│   ├── ot.py
│   ├── requirements.py
│   └── main.py
└── sets                      This directory contains the sets saved.
    ├── alice.txt
    └── bob.txt
```

The `code` folder contains all the file that we will utilize for the secure communication. `ot.py, util.py, yao.py` are utilities taken from Ojroques repo.

3.1.2 Coding part

First thing first, we need to represent the *circuit*; we decided to create the `add.json` file to code the circuit. Each wire is mapped to a number and linked with the other following the structure represented in Figure 4.
Each gate that appears in figure 4 is represented in a single JSON line.
The syntax is:

```
"id" :  <gateId>, "type" :  "<TYPE>", "in":  [<gateIdInput1>, <gateIdInput2>]
```

In this case the circuit is represented as follows:

```
{"name":"adder",
 "circuits":[
  {
    "id":"8 bit adder",
    "alice":[47,40,33,26,19,12,5,1],
    "bob":[48,41,34,27,20,13,6,2],
    "out":[62,50,43,36,29,22,15,8,3],
    "gates":[
        {"id":3,"type":"XOR","in":[1,2]},
        {"id":4,"type":"AND","in":[1,2]},

        {"id":7,"type":"XOR","in":[5,6]},
        {"id":8,"type":"XOR","in":[4,7]},
        {"id":9,"type":"AND","in":[4,7]},
        {"id":10,"type":"AND","in":[5,6]},
        {"id":11,"type":"OR","in":[9,10]},

        {"id":14,"type":"XOR","in":[12,13]},
        {"id":15,"type":"XOR","in":[11,14]},
        {"id":16,"type":"AND","in":[11,14]},
        {"id":17,"type":"AND","in":[12,13]},
        {"id":18,"type":"OR","in":[16,17]},

        {"id":21,"type":"XOR","in":[19,20]},
        {"id":22,"type":"XOR","in":[18,21]},
        {"id":23,"type":"AND","in":[18,21]},
        {"id":24,"type":"AND","in":[19,20]},
        {"id":25,"type":"OR","in":[23,24]},

```

```
            {"id":28,"type":"XOR","in":[26,27]},
            {"id":29,"type":"XOR","in":[25,28]},
            {"id":30,"type":"AND","in":[25,28]},
            {"id":31,"type":"AND","in":[26,27]},
            {"id":32,"type":"OR","in":[30,31]},

            {"id":35,"type":"XOR","in":[33,34]},
            {"id":36,"type":"XOR","in":[32,35]},
            {"id":37,"type":"AND","in":[32,35]},
            {"id":38,"type":"AND","in":[33,34]},
            {"id":39,"type":"OR","in":[37,38]},

            {"id":42,"type":"XOR","in":[40,41]},
            {"id":43,"type":"XOR","in":[39,42]},
            {"id":44,"type":"AND","in":[39,42]},
            {"id":45,"type":"AND","in":[40,41]},
            {"id":46,"type":"OR","in":[44,45]},

            {"id":49,"type":"XOR","in":[47,48]},
            {"id":50,"type":"XOR","in":[46,49]},
            {"id":51,"type":"AND","in":[46,49]},
            {"id":52,"type":"AND","in":[47,48]},
            {"id":53,"type":"OR","in":[51,52]},

            {"id":54,"type":"AND","in":[47,48]},
            {"id":55,"type":"NOT","in":[47]},
            {"id":56,"type":"NOT","in":[48]},
            {"id":57,"type":"AND","in":[55,56]},
            {"id":58,"type":"OR","in":[54,57]},
            {"id":59,"type":"AND","in":[47,58]},
            {"id":60,"type":"NOT","in":[58]},
            {"id":61,"type":"AND","in":[50,60]},
            {"id":62,"type":"OR","in":[59,61]}
        ]}]}
```

We then created `class Alice, class Bob` in `main.py` to represent their behaviour. It was decided that both parties have command line user interaction with the user, so we developed `askForInput` in `requirements.py`. In the same file we can also find two functions `saveSet, readSet` used to write in two external files `sets/alice.txt` and `sets/bob.txt` the input values; this will be useful when we want to check the correctness of the computed result, using `verifyOperation`.
Since the communication is encrypted, we decided to use *AES-CBC* as technique; we can see that in the file `yao.py`, more precisely in the two functions `encrypt, decrypt`.

As written in the code, we opted to prepend the IV at the cyphertext to avoid reusing the same IV for all the communications.
We can represent the pseudocode that represents the behaviour listed before.

Algorithm 1 Alice's behaviour

```
1: send(circuit)
2: aliceSet ← askForInput()
3: saveSet(aliceSet)
4:
5: aliceValue ← sum(aliceSet)
6: aliceE ← encrypt(aliceValue)
7:
8: r ← OT.getResult(aliceE, bKeys)
9:
10: result = decimal(r)
11: msg ← verifyOperation(result)
12: sendToBob(msg)
13: printResult(result, msg)
```

Algorithm 2 Bob's behaviour

```
1: receive(circuit)
2: bobSet ← askForInput()
3: saveSet(bobSet)
4:
5: bobValue ← sum(bobSet)
6:
7: OT.sendResult(bobValue)
8:
9: msg ← receive()
10: printResult(msg)
```

4 Usage

We will briefly explain how to install the program and how it works. In my repo there is also a graphical visualization of the executing program.

4.1 Installation

To run the program you need some packages:

- *ZeroMQ* used in `util.py` to implement communication;
- *SymPy* to use prime numbers;
- *AES* from encryption;
- *Bits* from bitstring, to do conversions from 2-complement to integers.

Simply run: `$ pip3 install pyzmq sympy cryptography bitstring`

4.2 Execution

As already writte, there are two parties: Alice and Bob. You must run on two different terminals them. From now, for simplicity, we assume that we are positioned inside `/src/`. From Alice's Terminal, simply run `make alice`, while for Bob's Terminal run

run `make bob`.
Now the connection will be established and the program will ask the user to give the input set, for each party. Once done, the result will be computed and printed in each terminal.

4.2.1 Example

A practical example will follow, we will show the output of each terminal.
Alice' side:

```
$ make alice
python3 code/main.py alice   -c circuits/add.json
Enter the list of integers of Alice's set: 1 2 -5
[1, 1, 1, 1, 1, 1, 1, 0]

======== 8 bit adder ========

Values of the computation

Syntax: parties/result:  [list of bits] = [correspective values]

Alice: [8, 7, 6, 5, 4, 3, 2, 1] = 1 1 1 1 1 1 1 0
Computed result by circuit: [53, 51, 46, 41, 36, 31, 26, 21, 17] = 1 1 1 1 1 1 0 0 0

The sum has been done correctly, result is -8.
```

Bob's side:

```
$ make bob
python3 code/main.py bob
Enter the list of integers of Bob's set: 4 6 -16
Received 8 bit adder
[1, 1, 1, 1, 1, 0, 1, 0]

Syntax:  parties/result  [list of bits] = [correspective values]

Bob [16, 15, 14, 13, 12, 11, 10, 9] = 1 1 1 1 1 0 1 0


The sum has been done correctly, result is -8.
```

Interpretation: As already written in the output, the syntax is
`parties/result:  [list of bits] = [correspective values]`.

The numbers on the left represent the bits in the circuit `add.json`, while the right ones represent the corrispective values. The last line is the output of the function `verifyOperation`.

References

[1] Moni Naor and Benny Pinkas. Computationally secure oblivious transfer. *Journal of Cryptology*, 18, 2005.

[2] Wikipedia contributors. Secure multi-party computation — Wikipedia, the free encyclopedia, 2023. [Online; accessed 10-May-2023].

[3] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982.