Maximum of a set of numbers secure multiparty computation using Yao's protocol documentation

Gabriele Castellani

June 22, 2024

1 Introduction

The aim of this project was to implement a two-party secure function evaluation, using Yao's protocol. The function implemented is the maximum of two set of values. The implementation is based on the GitHub repository `https://github.com/ojroques/garbled-circuit`. In the following sections all the details of the implementation are given.

2 How to

This section is subdivided in subsection that give all the necessary information to run correctly the project.

2.1 Dependency installation

This implementation is written in Python 3.10, all the test for the protocol have been run in Windows, here is a list of the necesary dependencies:

- PyZeroMQ: to initialize and use the local sockets
- SymPy: to manipulate and use prime numbers
- PyCryptoDome: for all the necessary encryptions and decryptions

It is possible to install these with pip one by one or also to run "pip install -r .\resources\requirements.txt" from command line from inside the **src** folder.

2.2 How to run

Since all the files are inside the **src** folder to run this implementation of the Yao's protocol from command line you have to reference that folder, so to run the project you can move inside the **src** or call it from outside, but paying attention to call the main from inside the **src** folder, so the command in the first case, where you already are inside the mentioned folder would be: "python main.py". In the other case, when called from outside its folder, the command would look something like: "python ...\garbled_circuit\src\main.py". The main.py file will then create two subprocesses one for alice and one for bob and each of them will execute its part of the protocol as explained in the "Protocol description" section. It is not possible to call alice and bob processes singularly with this implementation. As already mentioned to change the inputs of alice or bob it is necessary to modify the files Alice.txt or Bob.txt inside the **resources** folder, these files shall not be deleted or moved, shall not be empty(otherwise it will throw an error) and each input must be an integer greater than or equal to 0, separated from each other by a whitespace. All the outputs of the protocol are stored inside the **outputs** folder. The preliminary information that alice sends to bob are expressed following the format specified in the original library implementation.

3 Project structure

The original library has been refactored, mainly for the sake of legibility. For almost all the files now there is only a single class inside, with the exception of the file garbledCircuit.py, that contains the classes garbledCircuit and also garbledGate. The overall structure of directories can be then represented as follows:

```
src
├── circuits
├── outputs
├── resources
├── util
└── yao
```

Inside **src** there are all the other sub-folders and the files: alice.py, bob.py and main.py that implement, respectively, the functionality for the garbler(alice), the functionality for the evaluator(bob) and the main file orchestrates the flow of the whole protocol

In **outputs** all the meaningful outputs of the protocol get stored, such as: alice_preliminary_outputs.txt, that contains the information that alice sends to bob before the oblivious transfer takes place. The file final_result.txt contains the final output of the verification function of the main file and ot_intermediate_outputs.txt that contains all the meningful logged intermediate outputs of the oblivious transfer.

Inside **circuits** there is the total_circuit.json file, namely a json representing the max circuit for the protocol, that gets generated each time.

Inside **resources** there are the two files that the main uses to give the input to alice and to bob, namely Alice.txt and Bob.txt and a third file, requirements.txt which lists the necessary dependencies for the project

util contains some files used in the various procedures such as: socket.py and util.py.

Inside **yao** there are all the files that concern the yao protocol and the relative oblivious transfer, such as: evaluatorSocket.py, garblerSocket.py, primeGroup.py, ot.py, primeGroup.py, yao.py, yaoGarbler.py

4 Circuit description

The idea for the implementation of the circuit was to construct a n-bit compator recursively for a pair of numbers and iterate this process for all the inputs of alice and bob, using the output of each recursively contructed n-bit comparator for the next comparison in the iteration loop. The comparator circuit implementation is based on this resource found on the internet: "4-bit-magnitude-comparator" witch was useful for the task. The circuit created is different at every execution and it's able to deal with any bit-length representation and any number of inputs. The construction works as follows:

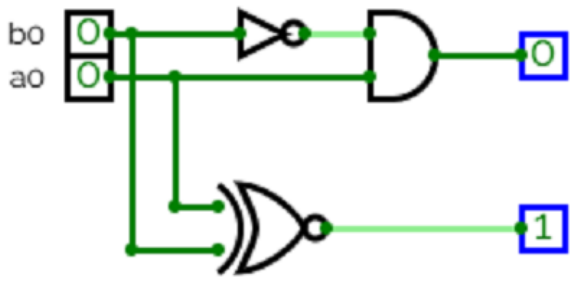

Figure 1: comparator of two binary numbers of length 1

This is the base case of the recursive procedure to generate all the necessary comparisons through the logic gates. a0 and b0 represent two binary numbers of length 1.

This circuit is a simple comparator in which the AND gate outputs 1 only if a0 is 1 and b0 is 0, so that means that a0 is greater than b0. The XNOR gate outputs 1 only if a0 and b0 are equal. These outputs are used to construct the n-bit version of this circuit:

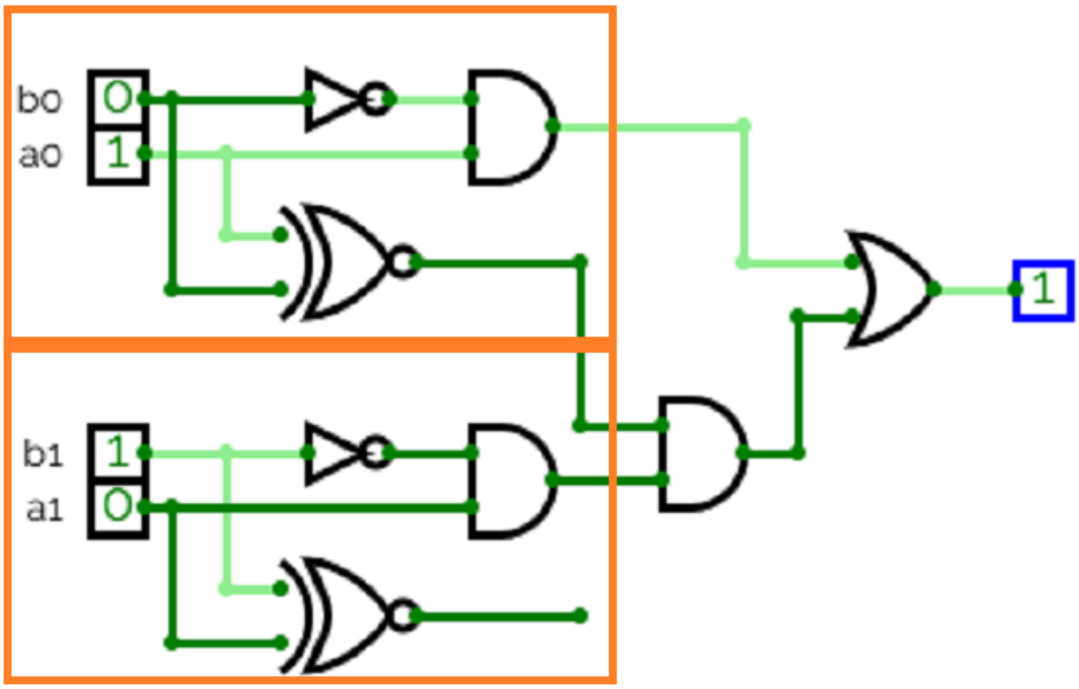

Figure 2: comparator of two binary numbers of length 2

The picture above shows a comparator of two binary numbers, both of length 2. The two rectangles areas enlighted represent each a comparator of two binary numbers of length 1.
The idea to construct the total comparator is to create a comprator for each pair of bits of the two numbers a and b, each of these with different magnitude. So first the circuit checks on the two most significant bits, namely if a0 is greater than b0 and, in that case, the output of the final OR gate will be 1, if these are equal the XNOR gate of the first comparator outputs 1 and this is used as a sort of carry that signals to consider the other less significant bits, so the comparator for a1 and b1 gets into play and if a1 is greater than b1 then again the final OR output will give 1 as result, and so on. Obviously if the final OR gate outpus 0, that means that b is greater or equal than a. This circuit construction works for any n-bit a and b.
The circuit is not complete yet, this construction is able to construct a comparator for only two numbers, to iterate this comparisons over all the inputs of alice and bob it

is necessay to create a choice mechanism that outputs the actual greater number, so that the circuit can then compare this local maximum with the next input(from alice or bob), given in the iteration loop. To do this a multiplexer circuit is used:

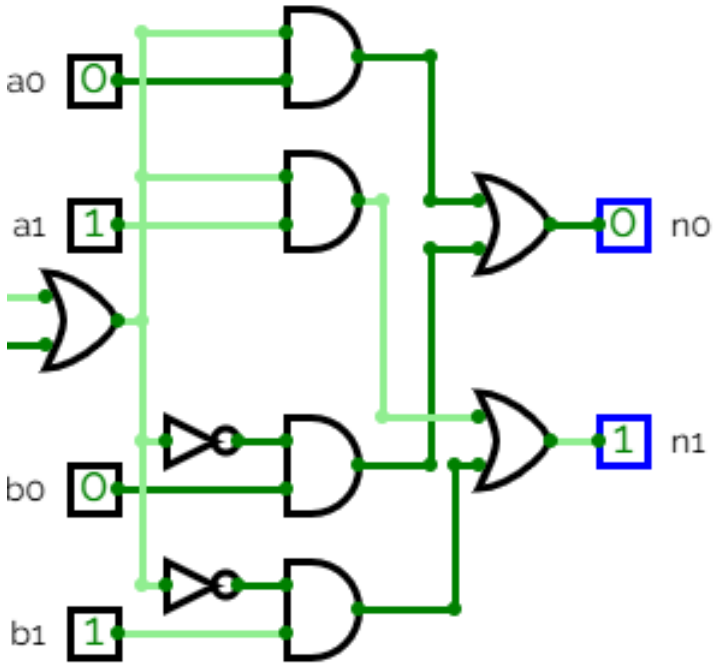

Figure 3: 2 bit multiplexer

The multiplexer is a standard circuit, its structure can b easily found *online* and in this project is used for selecting gates. In fact, based on the output of the OR gate of the comparator, this section of the circuit chooses the greater number between a and b and this result is the input for the next comparison with a different number from alice or from bob.

The max circuit so obtained for two n-bit numbers looks like this:

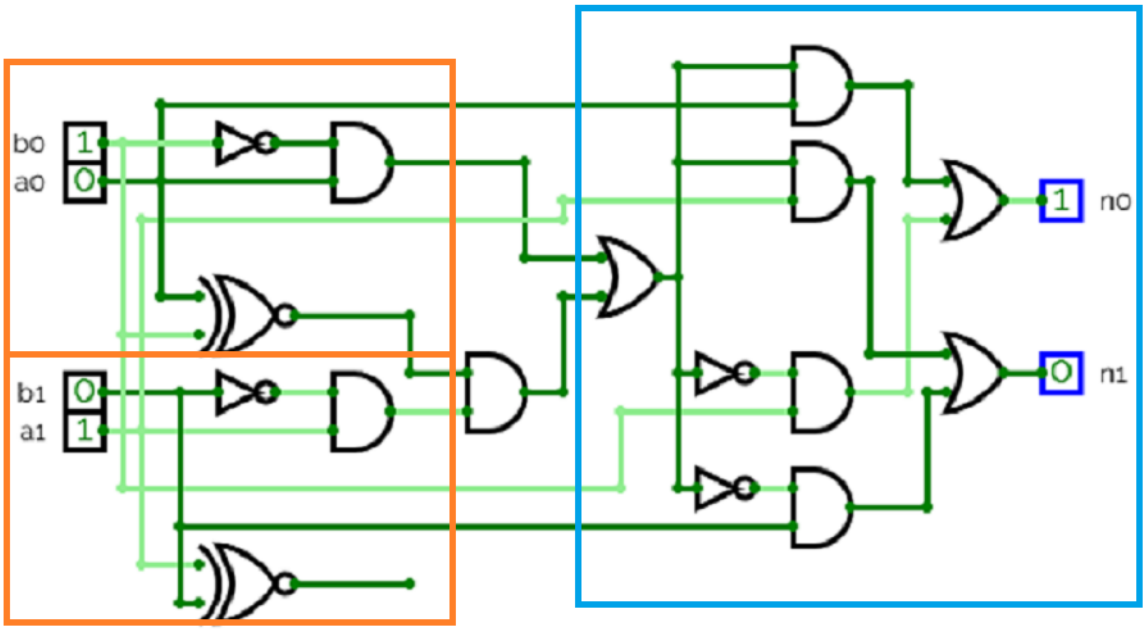

Figure 4: complete circuit

In this picture all the important sub-circuits are enlighted. Iterating this construction over all inputs of alice and bob, each comparison finds a local maximum and at the end the global maximum is the result.

5 Protocol description

In this section a description of the implementation of the Yao's protocol is given. The best way to represent this is with a pseudocode, actually one code for each party: alice and bob. Each step of this code will then be explored.

```
Data: alice_inputs
Result: the maximum of alice and bob's set of numbers
read_input(alice_inputs);
exchange_max_bit_length_and_number_of_inputs(input_length, max_bit_length);
circuit = create_max_circuit();
send_preliminary_information(circuit, garbled_tables, out_gates);
ot:{
    send_garbled_inputs_with_ot();
    send_suggestion_messages_with_ot();
    receive_bob_result();
}
```

Algorithm 1: Alice

```
Data: bob_inputs
Result: the maximum of alice and bob's set of numbers
read_input(bob_inputs);
exchange_max_bit_length_and_number_of_inputs(input_length, max_bit_length);
receive_preliminary_information();
ot:{
    receive_alice_inputs_with_ot();
    choose_correct_message_with_ot();
    evaluate_and_send_result_computed_with_message_with_ot();
}
```

Algorithm 2: Bob

This flow is orchestrated by the main.py file, alice and bob only deal with the implementation of the methods required by the main file to perform correctly the protocol. The inputs are read in the main function only from text files (Alice.txt, Bob.txt) located in the **resources** folder. The main function also takes care of storing all the meaningful results of the protocol in the correct files, exclusively in the folder **outputs**. It is not possible to read inputs from different sources than the files mentioned, you can only modify those, moreover the intermediate results and final outputs are stored exclusively in the **outputs** folder and a brief summary of the execution is printed to console.
After reading the input alice and bob agree on the input size and bit representation to use in the protocol, both party actually receive a randomly generated value for the input_length, which is greater or equal than the actual input_length, the bit_length data is instead sent clear. This is done to reduce information leaking, alice and bob should not know about each other's input_length.
The missing values in the inputs are replaced by zeroes to reach the agreed upon input_length, the inputs before beign sent are shuffled, to avoid sending multiple zeroes one after the other.

After this step alice produces the correct max-circuit and sends it to bob along with the garbled tables and the output gates, this is done by the method send_preliminary_information and this sent data is logged inside the file alice_preliminary_outputs.txt inside the **outputs** folder. Then the Oblivious Transfer can take place and this involves the exchange of some messages between alice and bob, where alice, at the request of Bob, sends a pair of encrypted messages as a suggestion to bob and bob, based on his input, chooses the correct message between the two and uses this information to evaluate.
Both parties communicate only through the Oblivious Transfer and to encrypt the data they use AES in CBC mode. The Oblivious Transfer intermediate results are logged by the main function in the file ot_intermediate_outputs.txt inside the **outputs** folder. After the Oblivious Transfer step is completed, bob sends the computed result, obtained with the Oblivious Transfer, to alice and the main file checks that the output of the function is the same as the output computed non multiparty way.
The result of this verification is stored in the file final_result.txt inside the **outputs** folder.