MPC: Find common elements in two sets of integer numbers

May 25, 2023

Contents

1 Introduction

The objective of this project is to create a script to evaluate the common elements between two parties, such that neither one of them knows about the inputs of the others and where an element is a 8-bit integer.

Given Alice's set: $AS = \{a_1, a_2, \ldots, a_n\}$ and Bob's set: $BS = \{b_1, b_2, \ldots, b_m\}$, our goal is for both Bob and Alice to know $RS = AS \cup BS$, without Bob knowing any of the elements $a_i \in AS$ and without Alice knowing any of the elements $b_j \in BS$.

2 The code

The code for this project was built around the garbled-circuit library.
Functionalities of the implemenetation:

- class Alice
  - **function "print"**: First Alice selects the circuit, generate the keys for the wires and a random value for each wire. Then Alice reads its input, and she exchanges with Bob the number of their inputs: N,M. At last Alice will send to Bob its input in a specific format: Given $a_1, a_2, \ldots, a_n$ the inputs of Alice, she sends M times a_1, M times a_2, ..., M times a_n.
- class Bob
  - **function "send_evaluation"**: Very similar to the print function of Alice, Bob first receives from Alice the circuit, then exchange with Alice the number of it's inputs, and finally computes the compare function. Given Bob's values: $b_1, b_2, \ldots, b_m$, Bob is expected from Alice to send for N times (number of inputs of Alice), all of it's inputs; so Bob will send b_1 then b_2 then ... then b_n then b_1, and it will repeat this N times.
- class ObliviousTransfer
  - **functions "get_result", "send_result"**: Alice and Bob perform oblivious transfer, a technique used to share with Bob only one of the two possible keys for Bob bound to a wire W.
- **classes GarbledCircuit and GarbledGate**: these classes are used to generate keys and random value associated with each wire W, aswell as to compute the circuit output, given the inputs.
- **functions "encrypt"/"decrypt"**: Both encryption and decryption is done with the Fernet library, which provides a "Mac and Encrypt" symmetric authenticated scheme, which uses AES with CBC-MODE with a 128 bits keys, and HMAC with a 128 bits key aswell (different from the one used for AES).

- **class Socket_connection_single_value**: create a temporary connection; used for both Alice and Bob to communicate the number of their inputs: N,M.
- **function "read_input_file"**: used to read the file containing the set of inputs of both Alice and Bob. Inputs are read as a list of integers: "1 10 251" are saved as $[1, 10, 251]$.
- **function "convert_input_list_to_nbit_list"**: Convert inputs from a list of integers, to a list of lists, where each inner list represent an integer in binary. Ex: Given $[1, 10, 251]$, the result of the conversion is: $[[0, 0, 0, 0, 0, 0, 0, 1], [0, 0, 0, 0, 1, 0, 1, 0]], [1, 1, 1, 1, 1,$

2.1 Python version and required libraries

- Python version: 3.10.9
- Library ZeroMQ for communications
- Library Fernet for encryption of garbled tables
- Library SymPy for prime number manipulation

2.2 Alice communication to Bob

```
inputs_of_alice ← read_inputs();
N ← length(inputs_of_alice);
send_to_Bob(N);
M ← receive_from_Bob_number_of_inputs()
for $i \leftarrow 0$ to $N$ do
    for $j \leftarrow 0$ to $M$ do
        #Perform OT
        c ← generate()
        send_to_Bob(c)
        $z_b$,$z_{b-1}$ ← get_from_Bob()
        if($z_b == z_{b-1}$): continue
        $r_0$,$r_1$ ← generate()
        $c_1 \leftarrow (g^{r_0}, \text{H}(z_b^{r_0} \oplus k_b^0))$
        $c_2 \leftarrow (g^{r_1}, \text{H}(z_b^{r_1} \oplus k_b^1))$
        send_to_Bob(c1,c2)
        #Send Alice input to Bob
        # Note that Alice's inputs are not sent in clear, instead Alice sends
          keys: Ex $k_a^0$ stands for 0 as input
        send_to_Bob(inputs_of_alice[i])
        output ← get_result_from_Bob()
    end
end
```

2.3 Bob communication to Alice

```
inputs_of_bob ← read_inputs()
M ← length(inputs_of_bob)
send_to_Alice(M)
N ← receive_from_Alice_number_of_inputs()
for $i \leftarrow 0$ to $N$ do
    for $j \leftarrow 0$ to $M$ do
        #Perform OT
        c ← receive_from_Alice()
        $k \leftarrow$ generate()
        $z_b$,$z_{b-1} \leftarrow g^k$, $\frac{c}{g^k}$
        $c_1$, $c_2$ ← receive_from_Alice()
        $z_b^{r_0} \leftarrow (g^{r_0})^k$
        $z_{b-1}^{r_1} \leftarrow (g^{r_1})^k$
        $k_b^0 \leftarrow H(z_b^{r_0}) \oplus (H(z_b^{r_0} \oplus k_b^0))$
        $k_b^1 \leftarrow H(z_{b-1}^{r_1}) \oplus (H(z_{b-1}^{r_1} \oplus k_b^1))$
        #Bob has access to both keys for Bob, without Alice knowing which
         Bob will use
        alice_input ← get_alice_input()
        output ← compute_output(alice_input, inputs_of_bob[j], $k_b^0$, $k_b^1$)
        send_output_to_alice(output)
    end
end
```

3 The Compare circuit

The compare circuits is a boolean circuit built using logical doors and in the code (circuits/cmp.json) is saved the compare circuit in a json form.
In the following sections is shown how the circuit used in the implementation described above was built and how it works, moreover it's also shown how to create a compare circuit for a general n-bit integer.

3.1 The 8-bit compare circuit

Given two 8 bit values: A= $a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8$, B= $b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8$, the 8-bit compare logical circuit 3.1 aim is to return OUT= 1 if A=B, else OUT= 0.
Starting from the end of the circuit, we can see that OUT= 1 iff every "subunit" on the left outputs 1. Each subunit is composed by 2 inputs: a_i, b_i, a XOR gate and a NOT gate.
The i-th subunit returns 1 iff $a_i = b_i = 0$ or $a_i = b_i = 1$ 3.1, hence OUT will be equal to 1 only if A=B.

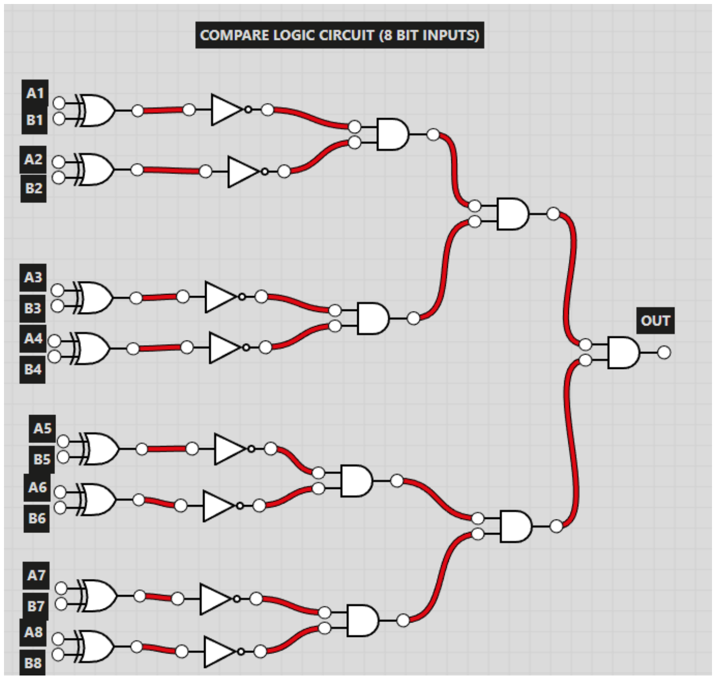
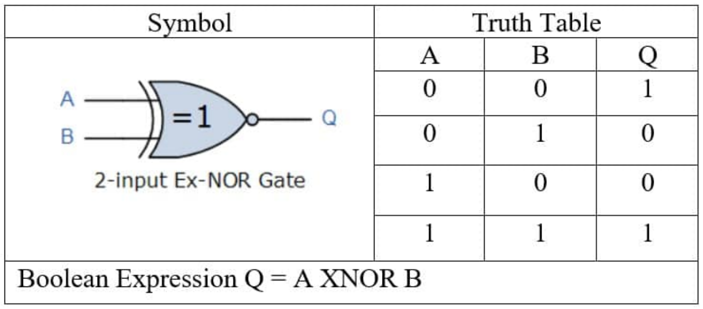


Figure 1: 8-bit compare logical circuit, built using logic.vy



Symbol	Truth Table		
	A	B	Q
	0	0	1
	0	1	0
	1	0	0
	1	1	1
Boolean Expression Q = A XNOR B			

A
B
=1
Q
2-input Ex-NOR Gate

Figure 2: XNOR truth table

3.2 How to create an n-bit compare circuit

To build a n-bit circuit for compare it is sufficient to use ”subunits” described earlier 3.1.
Given A= $a_1, a_2, \ldots, a_n$ and B= $b_1, b_2, \ldots, b_n$:

1. Link each i-th couple a_i, b_i to a XOR and then NOT (or a XNOR) gate; we will call these ”subunits”, and we obtain a set $s_1, s_2, \ldots, s_k$, where s_i is a subunit. If a single gate remains out add it into the set of couples of subunits as a single subunit;

2. Subdivide the subunits into couples $(s_1, s_2), (s_3, s_4), \ldots, (s_{k-1}, s_k)$, and link each

couple to an AND gate.

3. Keep going subdividing into couples and linking them to AND doors until you have no more couples.

In the end you will end up with a logic circuit for compare for n-bits.

4 Running the code

4.1 Insert inputs

You can insert inputs in the files inside the folder ”inputs/”, that are named after both Alice and Bob, which contains respectively the set of integers of Alice and the set of integers of Bob which have to be compared.

Inputs are integer numbers $\in [0, 255]$ interspaced by a blank space.
Example: ”65 1 0 155 32 255 1 3 1 0 83” (without ””)

4.2 Run the script

1. be sure that ports 4080, 40800 and 40801 aren't currently occupied;
2. open the terminal for Bob and enter an enviroment with the required libraries 2.1;
3. execute Bob: ”python main.py bob” or ”./main.py bob”, now Bob will keep listening for requests from Alice;
4. open the terminal for Alice and enter an enviroment with the required libraries 2.1;
5. execute Alice: ”python main.py alice -c circuits/cmp.json” or ”./main.py alice -c circuits/cmp.json”;
6. if you want to run the program with different inputs, it's sufficient you insert the inputs and re-run Alice, you don't need to restart Bob.

At the end of the computation both Bob and Alice will print the common set of elements.

5 Conclusions

This project accomplished it's target objective: creating a working MPC enviroment for the ”compare” function between sets of 8-bit integer numbers (chapter 2).
In chapter 3 it is shown how to create a general version of the compare circuit that

could extend the current integer range $[0, 255]$, to the range $[0, 2^i]$ for a i-bits compare circuit, and finally in chapter 4 we showed how to run the code and insert inputs.

A possible future development is to make this code also work for 8 bits floating point numbers. In order to do so the compare circuit would remain the same, but we would need to reconognize if the input that was received was an integer or a float (we could write a lexer/parser couple), and then to process them accordingly.