Introduction to Cybersecurity Project Report: Yao's Garbled Circuits

Alessio Boiardi

June 14, 2024

1. Python Environment

The project has been developed using Python version 3.11.8 in a virtual environment using Conda on Windows 11. All the packages from the environment can be found in the standard format "requirements.txt" file included in the archive for the project and can be installed with *pip install –r requirements.txt*. Some extra unused packages may have been included in the requirements file due to their use during prototyping and testing. I decided not to remove them to ensure compatibility.

2. Running the Script

The core of the program resides in the "format.py" file. I decided not to change its name to ensure compliance with the requirements. This file calls the others programmatically so no direct interaction with those is required from the user.

To run the script in its default form it is sufficient to call "python format.py". Doing so will run the **MAX** circuit over the default inputs of 4 bits length and set size equal to 5. The corresponding circuit has already been pre-computed and can be found inside the "circuits" directory in the project's files.

Additional parameters can be passed to the script:

- *--set-size, -s*: An integer representing the size of the set of numbers for each party. The default value of 5 means that the function will run over 10 values, 5 from Alice and 5 from Bob.
- *--alice, -a*: The path to the input file for Alice's values. Default is "Alice.txt" (in the same directory as the script). Case sensitive.
- *--bob, -b*: The path to the input file for Bob's values. Default is "Bob.txt" (in the same directory as the script). Case sensitive.

In addition to calling the script, it's necessary to provide the input files for both Alice and Bob. As described above, the default values point to two similarly named files in the same directory as the script: "Alice.txt" and "Bob.txt". Included in the archive are two sample files containing 16 numbers each, i.e. all the possible 4-bit numbers from 0 to 15. The format of the file is very lenient: The file will be read correctly as long as it exists. Any 1 or 0 encountered is added to

the input and everything else gets discarded. That being said, the recommended input format for legibility reasons is the one provided in the sample file. When calling the script with a different *–set-size* value, the user should take care to change the input files accordingly. The script is set to never fail on a shorter/longer input file unless the number of bits contained wherein is not a multiple of 4. If the file contains more/less numbers than the *–set-size* parameter, it will be appended with 0s or truncated respectively to the expected input size. Appending 0s does not change the outcome of the script because it is already the lowest possible number.

3. Implemented Function

The function I chose to implement is the function that computes the **MAX** of two sets of values, as proposed in the project assignment. This function will assume that Alice and Bob have two sets of values of equal size and wish to find out which is the maximum between them through secure *MPC* with *Yao's Garbled Circuits* and *Oblivious Transfer*. A careful reader might note that to achieve this, Alice and Bob would need to agree on a set size beforehand through a different channel. However, since the script is set to automatically pad any shorter sets, this is not seen as a limitation because the agreed-upon set size would only function as a **maximum** set size and both Alice and Bob are free to choose a lower set size without the other party ever knowing.

4. Circuit Description

The circuit implementation was done through a script that translates a boolean function into a circuit with the appropriate format defined by the original library[1] provided in the project assignment. The script is named "boolean.py" and gets called at runtime by the core script if the circuit for the desired set size hasn't been generated yet.

The circuit's design is made up of blocks cascaded into one another. The basic block is a comparator that allows us to compare two 4-bit numbers to see which one is greater. Since the full truth table for such a system would have been too big (two 4-bit inputs would amount to a table with 2^8 = 256 rows) to compute by hand, I employed a "divide and conquer" approach as suggested on this website[2] to simplify it and obtained the following significance table (also from the same website).

[1] https://github.com/ojroques/garbled-circuit
[2] https://technobyte.org/2-bit-4-bit-comparator/

A3B3	A2B2	A1B1	A0B0	A>B	A<B	A=B
A3>B3	x	x	x	1	0	0
A3<B3	x	x	x	0	1	0
A3=B3	A2>B2	x	x	1	0	0
A3=B3	A2<B2	x	x	0	1	0
A3=B3	A2=B2	A1>B1	x	1	0	0
A3=B3	A2=B2	A1<B1	x	0	1	0
A3=B3	A2=B2	A1=B1	A0>B0	1	0	0
A3=B3	A2=B2	A1=B1	A0<B0	0	1	0
A3=B3	A2=B2	A1=B1	A0=B0	0	0	1

With this approach, we can break down the truth table into chunks and resort to only compare the most significant bits that aren't the same, then keep going down until we find one that is higher or lower. If all bits are the same, then the two numbers are equal. In our case, the penultimate column is redundant (as it simply is the negation of the previous column *OR* the following column).

We then need to find the basic functions to translate these blocks into circuits:

- **A=B** can be expressed with the *NXOR* gate (*NOT EXCLUSIVE OR)* which has the following truth table:

  A B C
  0 0 1
  0 1 0
  1 0 0
  1 1 1

  Clearly, the output will be 1 when both inputs are the same. We call this function **X** for future use. We will refer to this function applied to the respective bits as **X3**, then **X2**, **X1** and **X0**.
  Therefore, we can already check if the two values are equal through:
  Z = X3 AND X2 AND X1 AND X0
  However, the library doesn't allow for gates with more than two inputs, so we must nest these comparisons as such:
  Z = AND(X3, AND(X2, AND(X1, X0)))

- **A > B** is a compound expression that can be true in multiple cases, so we need to collect these cases and *OR* them together, running down the table we find (*NBX* means that we negate the corresponding bit of *B*):
  - Case 1: **A3 > B3** -> **Y3=AND(A3, NB3)**
  - Case 2: **A3=B3**, **A2 > B2** -> **Y2=AND(X3, AND(A2, NB2))**
  - Case 3: **A3=B3**, **A2=B2**, **A1 > B1** -> **Y1=AND(X3, AND(X2, AND(A1, NB1)))**
  - Case 3: **A3=B3**, **A2=B2**, **A1=B1**, **A0 > B0** -> **Y0=AND(X3, AND(X2, AND(X1, AND(A0, NB0))))**

We combine these together with an *OR* expression that we call **Y**.

Finally, we know the values for **Z** and **Y** and we *OR* them together since we don't care which value we return when they are equal, so we consider this the same as when **A** is greater than **B**. The only thing left to do is output the full value instead of just the greatness indicator. To do so, we compute the following bits (where *NY* indicates the negation of *Y*):

- **Z3=OR(A3, B3)**
- **Z2=OR(AND(Y, A2), AND(NY, B2))**
- **Z1=OR(AND(Y, A1), AND(NY, B1))**
- **Z0=OR(AND(Y, A0), AND(NY, B0))**

The output of this circuit is (**Z3**, **Z2**, **Z1**, **Z0**), i.e. the higher value of the two.

Once we have this circuit in place, all we need to do is cascade its output as input into another identical circuit together with the next value from one of the sets. We do this programmatically with a for loop and a generator for the gate IDs to allow circuits of arbitrary length to be constructed. In total, we perform as many comparisons as there are elements in the set minus 1 so our algorithm is **O(n)**, which is provenly the optimum for finding the maximum value in an unordered set.

5. Protocol Flow

The flow of information between Alice and Bob for the MPC and OT hasn't been modified from what was already in place in the original library. A brief description follows:

1. Bob starts and puts itself into active listening on a local socket.
2. Alice starts and reads the input circuit from the file.
3. Alice generates the garbled circuit, which involves generating random p-bits, encryption keys and using these to garble the circuit.
4. Next, Alice sends Bob the information about the circuit, the garbled tables and the p-bits of the output wires. No Oblivious Transfer occurs here as this is public knowledge.
5. Alice then sends Bob her encrypted inputs.
6. After Bob receives Alice's inputs, the Oblivious Transfer protocol begins.
7. For each of Bob's input wires, the following Oblivious Transfer loop executes:
    a. Bob sends Alice the ID of the input wire on which to perform the OT.
    b. Alice receives such ID.
    c. Bob starts the oblivious transfer protocol.
    d. Alice starts the oblivious transfer protocol.
    e. Alice sends Bob the public key and random messages.

f. Bob picks one of the messages and sends its encryption with a random *k* back to Alice.
g. Alice decrypts the encrypted value with both messages and gets back two candidates for *k*.
h. Alice encrypts both secrets with *k1* and *k2* respectively and sends them back to Bob.
i. Bob uses *k* to decrypt the correct secret.

8. Finally, Bob evaluates the complete circuit and sends the result back to Alice.

Algorithm 1 Protocol Flow Between Alice And Bob

1: **start:**
2: start Bob
3: start $Alice$
4: $circuit_{Alice} \leftarrow file$
5: $garbled\_circuit_{Alice} \leftarrow \text{YaoGarbledCircuit}(circuit)$
6: $Bob \leftarrow (circuit_{Alice}, garbled\_tables_{Alice}, pbits^{out}{}_{Alice})$
7: $Bob \leftarrow encr\_input_{Alice}$
8: **loop** for $wire$ in $input_{Bob}$:
9: $\quad Alice \leftarrow gateID_{Bob}$
10: $\quad Bob \leftarrow \text{OT}(key_{ID,0}, key_{ID,1})$
11: $result_{Bob} \leftarrow \text{YaoEvaluate}(garbled\_circuit)$
12: $Alice \leftarrow result_{Bob}$

6. Additional Information

Code Ownership

All the code except for that which was already present in the original library has been written by me. Significant modifications to the original library have been made in the file "main.py". Other files from the library have only been modified in what they output to console for better legibility.

Additional libraries used are:

- *json*: To save the circuits and the exchange of information between Alice and Bob to json files.
- *os*: To work with filepaths.
- *threading*: To start Bob's thread in the same process as Alice.

Notes on Efficiency and Security

As it is written, the program aims to compute the max value between both sets all within a single circuit because the requirements of the project stated thus.

This is suboptimal since it makes no sense for Alice and Bob to send each other all their values if they already know that all but one are potential candidates for max. The ideal way to compute this would be that each party locally identifies their max and then performs MPC over that alone. Not only would this design be more lightweight, providing more efficiency both in terms of development effort and computation gains, but it would also be more secure. In the interest of security, any process should aim to minimize the transfer of sensitive information, never mind how supposedly secure the transfer channel and algorithms are. Should these algorithms fail or leak information in any way, sending the whole set to the other party means endangering more data than sending just one value.