**622.755 – Introduction to Cybersecurity** Winter Term, 2020/21

# Implementation Documentation

Anna Nava 12109889

## 1 Introduction

The implemented program is an example of a simple Yao protocol for two parties with AES. Briefly, **what is Yao's protocol for two party computation?** It's a cryptographic protocol that enables two parties to jointly compute a function on their inputs, keeping their inputs secret to the other party.
Let's call the parties Alice (the *garbler*) and Bob (the *evaluator*). Here the principal steps of the protocol:

1. Express the function as a boolean circuit.
2. Alice builds the *garbled circuit*, encrypting the circuit.
3. Bob receives from Alice the garbled circuits and her encrypted inputs.
4. OT (oblivious transfer) step: Bob needs to encrypt his inputs too, but he doesn't know how to encrypt. Thanks to the OT, Bob obtains his encrypted inputs with Alice's help, but both Bob and Alice don't reveal any other information to the other party.
5. Bob evaluates the circuits.
6. Bob and Alice share the output.

## 2 Function and circuit

I decided to implement the sum of a set of values.
My implementation is able to process integers of size at most 4-bits. This means that given $a = \sum a_i$, where $a_i$ are Alice's inputs, and $b = \sum b_i$, where $b_i$ are Bob's inputs, then $a, b \in I = [0, 15]$.
I considered a 4-bit full adder circuit. A **full adder** is simply a circuit which adds binary numbers. The easiest example is a 1-bit full adder: in this case, the goal is to add two one-bit numbers $a_0$ and $b_0$: to do that, also a third bit $c$ always equal to 0 is considered (the initial *carry*). It's necessary, because $a_0 + b_0$ can also be equal to a 2-bit number. In particular, the circuit will follow this easy computation: $S = a_0 \oplus b_0 \oplus c$ and $C_{out} = (a_0 \wedge b_0) \vee [(a_0 \oplus b_0) \wedge c]$, where the sum will be equal to $C_{out}\ S$.
The 4-bit full adder is obtained in the same way, concatenating four 1-bit full adder. The circuit is illustrated in the picture below, where:

- Alice's input is a 5-bit number:

$$c\ a_3\ a_2\ a_1\ a_0,$$

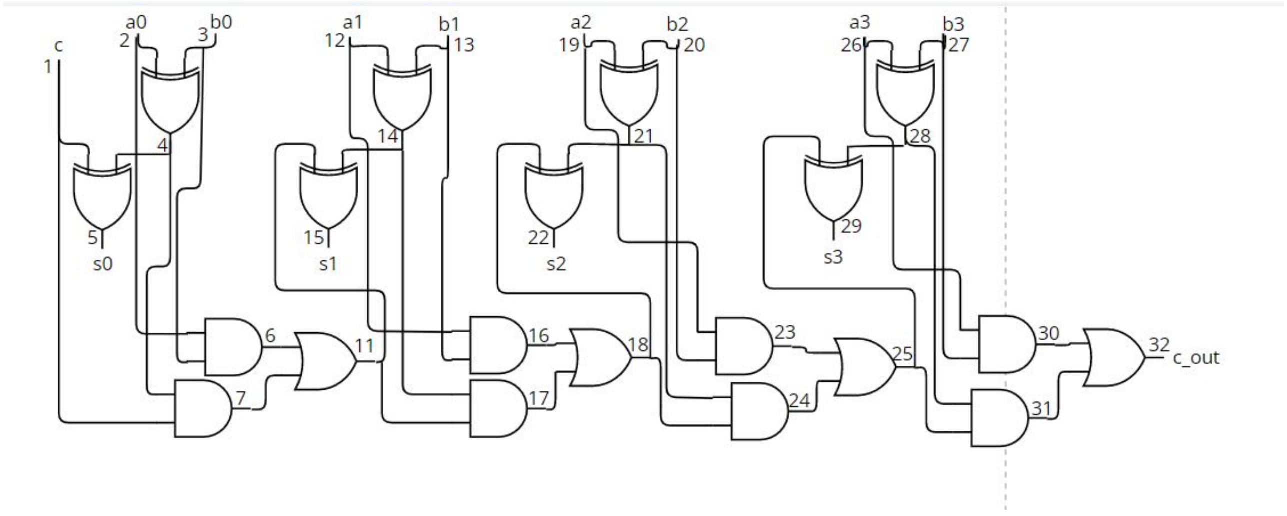
  where $c$ is the initial carry and is always equal to zero

- Bob's input is a 4-bit number:

$$b_3\ b_2\ b_1\ b_0$$

- The final sum is a 5-bit number:

$$c_{out}\ s_3\ s_2\ s_1\ s_0$$



## 3 Implementation

I used the library [3], in particular the unmodified `yao.py` and `util.py` and the modified `main.py`. The additional packages `pandas` and `re` are needed. Note that in `yao.py` the function utilized for encryption and decryption is Fernet, which uses AES in CBC mode with a 128-bit key [1].
A note about the notation: the parameters which in the code are called *p-bits* are the (secret) signal bits for each wire.
Let's see how I modified `main.py`: first of all, I decided to use the class `LocalTest` (and for this reason I removed classes `Bob` and `Alice` from the code). The disadvantage of this choice is the fact that the communication between Alice and Bob is not really clear, but I added a pseudocode about that.
Immediately after reading Alice's input from the text file `Alice_inputs`, Alice calculates the sum of her set of values and it will be one of the inputs for the circuit; the same

happens with Bob's input. In fact, it's faster to calculate immediately the sum of each input and then apply the protocol; moreover, there isn't any danger for the secrecy of the inputs because neither of the parties is sharing information in this way.
I added five more functions in the class:

1. `read_input`, which reads the file text with the set of values in input (firstly of Alice, later of Bob). If it is not a list of integers, it will print an error and return a string: in this way, soon later the code will be stopped by an error.

2. `sum_bin`, which simply calculates the sum of the values in input, inserted by the same user. If the sum is too big for the algorithm, this function will print an error and return a string: in this way, soon later the code will be stopped by an error.

3. `create_file`, which creates a `.csv` file with Alice's message to Bob. It's a table which contains Alice's encrypted inputs.

4. `bin_to_dec`, which simply converts a binary number to a decimal number.

5. `verify`, which is the function that verifies the correctness of the computations done using Yao's protocol.

## 4 Pseudocode: communication between parties

Since in the localTest class these passages are not explicit, it has been requested to do a pseudocode about the communications between Alice and Bob (I've chosen to use 1-2 Oblivious Transfer in this code, but it's possible to choose another OT):

**Alice's side**

```
# Alice constructs b_keys
b_keys := dict mapping each Bob's wire to a pair (key,encr_bit)

send Alice's encrypted inputs to Bob

# 1-2 Oblivious Transfer
for each element in b_keys:
    receive gate ID {wire} from Bob, where to perform OT

    generate RSA keys ((d,p,q),(N,e))
    send public keys (N,e) to Bob
    generate two random messages ($msg_0$,$msg_1$)
    send ($msg_0$,$msg_1$) to Bob

    receive encrypted message m_bob from Bob
    # decrypt the message: the right decrypted one is $par_0$ or $par_1$
```

```
        $par_0$ := $(m\_bob - msg_0)^d$ mod N
        $par_1$ := $(m\_bob - msg_1)^d$ mod N
        # create new messages associated to the pairs b_keys[wire]
        $msg'_0$:=b_keys[wire][0]+$par_0$
        $msg'_1$:=b_keys[wire][1]+$par_1$
        send ($msg'_0$,$msg'_1$) to Bob
    # end OT

    receive the circuit evaluation sent by Bob
```

**Bob's side**

```
    b_inputs := dict mapping each wire of Bob to Bob's (clear) inputs

    receive Alice's encrypted inputs
    # initiate map from Bob's wires to (key, encr_bit) inputs
    b_inputs_encr := {}

    for each element (wire,b_input) in b_inputs:
        send gate ID {wire} to Alice

        # 1-2 Oblivious Transfer
        receive (N,e) and ($msg_0$,$msg_1$) from Alice

        generate random value $par$
        m_bob:= $(msg_{b\_input} + par^e)$ mod N
        send m_bob to Alice

        receive ($msg'_0$,$msg'_1$) from Alice
        # select the message which corresponds to {b_input}
        b_inputs_encr[wire] := $msg'_{b\_input} - par$
    # end Ot

    evaluate the result of the circuit
    send the evaluation to Alice
```

## 5 Script execution

Disclaimer: the instructions are for Windows, but it's easy to translate them for the other OSs.
To run the script, open the shell and insert the directory of the project folder:

```
cd path_project_folder
```

Then, insert the following command:

```
python3 main.py local -c circuits/sum.json
```

where `sum.json` is the file json which contains the circuit for the 4-bit full adder.
It's possible to insert the desired Alice's and Bob's inputs in the files `Alice_inputs.txt` and `Bob_inputs.txt`, which are in the same folder of the rest of the python files.

## 6 Outputs

In the end, the printed outputs on the shell are the sum of the values calculated with Yao's Protocol and the result of the check of the correctness of this sum.
Two files are also created: `file_message_alice.csv`, which contains Alice's message to Bob, and `output_result.txt`, which contains Bob's message to Alice (i.e. the result of the sum) and the result of the function which verifies the correctness of the MPC computation.

## References

[1] cryptography.io, Fernet, *https://cryptography.io/en/latest/fernet/#using-passwords-with-fernet*, JAN 2022

[2] Peter Snyder, *Yao's Garbled Circuits: Recent Directions And Implementations*, 2014

[3] Olivier Roques, garbled-circuit, *https://github.com/ojroques/garbled-circuit*, JAN 2022

[4] Wikipedia, 4-bit full adder, *https://en.wikipedia.org/wiki/Adder_(electronics)*, JAN 2022

[5] Wikipedia, Garbled circuit, *https://en.wikipedia.org/wiki/Garbled_circuit*, JAN 2022

[6] Wikipedia, Oblivious Transfer, *https://en.wikipedia.org/wiki/Oblivious_transfer*, JAN 2022