Technical report on Private Multi Party Computation using Yao's protocol

Lorenzo Bazzana
12245823
lobazzana@edu.aau.at

May 2023

1 Introduction

This report is aimed to describe the technical fuctionalities of an implementation of Yao's protocol [2] for private *Multi Party Computation* (MPC for short), to compute functions using a set of data owned by different parties, while maintaining privacy of the data itself.

The implementation presented is based on the library repository *garbled-circuit* by Olivier Roques and Emmanuelle Risson (publicly available on GitHub), but extends it in order to accommodate the specific problem at hand, described in the following.

1.1 Problem description

This implementation of Yao's protocol is specifically designed to calculate the maximum value in a set of 32-bit integers, partitioned between two parties, called "Alice" and "Bob". Following the protocol, Alice is the "garbler" (the party that generates the circuit and the garbled gates to be sent) and Bob acts as the "evaluator" (the party that extracts the correct values of the garbled tables and communicates them to the garbler). At the end of the protocol both parties know the maximum value, while none of the other values is exposed.

2 Computation and communication description

The idea behind the computation is that, since the implemented function must calculate the maximum of two sets of values, each party can compute the max of their own values locally, while the global maximum can be computed using Yao's garbled circuit.

2.1 Local computation

The local computation is exactly the same between both parties; the pseudocode of the local maximum computation is described by algorithm 1.

2.2 Alice's communication to Bob

Alice is the circuit generator, meaning that she must create the garbled tables that are later evaluated by Bob; she must also communicate the circuit to Bob beforehand. Since the circuit is dynamically generated based on the required number of bits (as described in section 3), there is a brief exchange with Bob regarding the size of the circuit. After that, the actual Yao's protocol exchange is started.

Alice's communication to Bob is described by algorithm 2.

Algorithm 1: Algorithm used to compute the local maximum

```
function local_max():
    input_list ←array(user_inputs())
    filter_invalid_inputs(input_list)
    max_value ← max(input_list)
    if max_value ≥ −2^31 and max_value ≤ 2^31 − 1 then
        return max(input_list)
    else
        error: "Integer out of range"
    end
```

Algorithm 2: Pseudocode of Alice's communication to Bob

```
alice_max ← local_max()
circuit ← generate_circuit(32-bits)
garbled_circuit ← garble(circuit)
send(bob, garbled_circuit)
binary_input ← convert_to_binary(alice_max)
encrypted_input ← encrypt(binary_input)
bob_keys ← generate_keys()
send(ot, encrypted_input, bob_keys)
circuit_result ← receive(ot)
verify_result(circuit_result)
```

2.3 Bob's communication to Alice

Bob is the party that must evaluate the circuit, given the garbled tables and the secret keys relative to his own inputs; these keys are obtained via the *Oblivious Transfer* (OT) protocol. Bob must also communicate to Alice the required bits needed to represent his local maximum before Yao's protocol can actually take place. Algorithm 3 explains Bob's communication to Alice.

Algorithm 3: Pseudocode of Bob's communication to Alice

```
bob_max ← local_max()
garbled_circuit ← receive(alice)
binary_input ← convert_to_binary(bob_max)
send(ot, binary_input)
bob_keys ← receive(ot)
circuit_result ← evaluate(garbled_circuit, bob_keys)
send(ot, circuit_result)
verify_result(circuit_result)
```

2.4 Security of the protocol

The key points that must be considered when analyzing Yao's protocol are *correctness* and *privacy*.

Correctness is the property of the protocol to compute the desired function without errors. This is correctly achieved if Alice's and Bob's max values are in the interval $[-2^{31}, 2^{31} - 1]$; if either Alice's or Bob's max is out of this range, they abort, because the value cannot be correctly represented. At

the end of the communication, both Alice and Bob know the result of the computation.

Privacy regards what one party can infer about the values of the other party from the protocol. By nature, the *max* function does reveal to a party that all the values owned by the other party are bounded by the value returned by the function. In the case of a malicious adversary, this can be exploited to learn more information, but in the case of a semi-honest adversary (which is the case that was treated in class), nothing more is learned, since the protocol does not expose anything about the actual number of bits required to represent Alice's and Bob's values.

3 Circuit description

The circuit used to compute the maximum between Alice's and Bob's local maximum values is based on the comparison using the difference of two values and using the sign of that difference to determine which one was greater. In order to simplify the circuit, the two inputs are actually A (Alice's max) and $-B$ (the opposite of Bob's max, calculated via software); including in the circuit the computation of $-B$ would have in fact added unneeded complexity, especially for the increase of AND gates [1]. The idea is schematized in algorithm 4:

Algorithm 4: Computation of the max between two integers

```
1 function circuit_max(A: int, B:int):
2 |  C ← A + (−B)
3 |  if C < 0 then
4 |  |  return −B
5 |  else
6 |  |  return A
7 |  end
```

Since determining the sign of an integer (in this case $C = A - B$) is solely dependent on the most significant bit of that integer, instead of actually computing all of C's bits, it is possible to just compute its most significant bit. This can be done by computing just the carry of the sum of all other bits and adding that to the sum of A's and $-B$ most significant bits.

The selection of the output is done by using a simple multiplexer that uses the sign bit to select either A or $-B$. The multiplexing bit is also one of the values returned by the protocol, and is used to determine whether the output of the circuit must be inverted: if it is 0, it means that the received value is A, and nothing else is required; if it is 1, we received $-B$, and thus we must first convert it to its opposite before printing it.

In order to deal with edge cases like sum overflow, the circuit is actually generated with a size of 33 bits: this is because the sum between two n-bit numbers can at most be an $(n+1)$-bit number, so we need a circuit size that is one bit bigger than the size of A and B. This also allows the circuit to work with negative integers.

Figure 1 illustrates an 8-bit version of the circuit.

4 Functionalities and project structure

The project is structured in different python files, each containing specialized classes and functions for a specific role in the protocol:

- `mpc.py`: the main file for the execution of the two parties.
- `alice.py`: the file containing Alice's class and the functions used by the garbler.
- `bob.py`: the file containing Bob's class and the functions used by the evaluator.

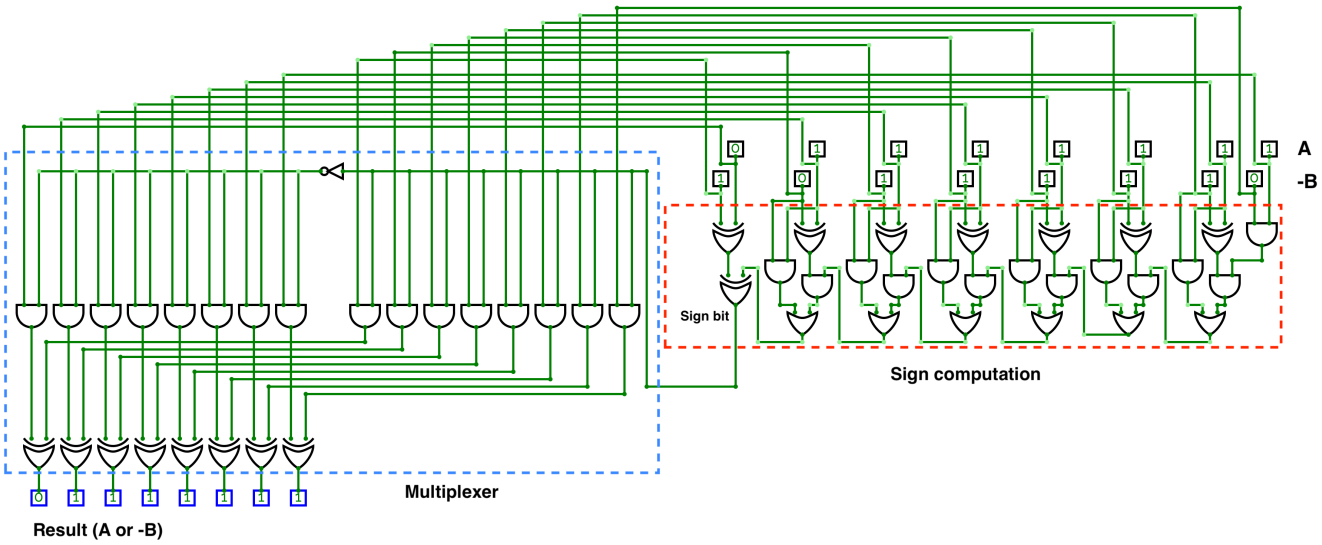

Figure 1: 8-bit version of the circuit used to compute the maximum. The actual 32-bit circuit can be derived by extending the subcircuit for the sign computation and the multiplexer.

- `garbler.py`: this file contains the class "YaoGarbler", from which Alice and Bob inherit. It is taken from the GitHub repository.
- `circuit_generator.py`: it contains the function used to generate the circuit. Since for each pair of Alice's and Bob's bits (a_i, b_i) the gates required to compute their sum and carry form a repeating pattern, arbitrarily extending the circuit is fairly easy, but it was decided to use a fixed size in order to make the computation more secure.
- `ot.py`: in this file there are the necessary functions and classes required to perform the OT. It is the same as the one on the GitHub repository, with a small modification: now the function `send_result` also returns the result to Bob, so that he too knows the output and can print it.
- `yao.py`: this file implements the evaluation part of Yao's protocol. It is the same as the one on the GitHub repository.
- `util.py`: this file contains utility functions and classes used to simplify and add readability to other classes' functions. It is mostly the same as the file in the GitHub repository, with the addition of the following helper functions:
  - `bindigits`: this function takes two integers n_1 and n_2 and returns the n_2-bit binary representation in two's complement of n_1.
  - `convert_result`: this function takes a python dictionary representing the result of the circuit evaluation and returns the relative binary and integer representation.

– `verify_computation`: this function verifies the correctness of Yao's protocol's computation, taking its result and comparing it to the maximum computed in a classical way: this is done by reading the contents of two files where Alice and Bob each write their own local maximums, and the global maximum is computed from those two values.

5 Script usage

5.1 Implementative and running details

The implementation is done using Python 3.10.9 and requires the following additional python libraries in order to correctly work:

- bitstring 4.0.2
- cryptography 39.0.2
- pyzmq 25.0.0
- sympy 1.11.1

Of all these libraries, "bitstring" is an addition specific to this implementation, while all the other packages are required to run the standard version of the GitHub library. The number next to each library indicates the version it was tested with, and it is highly recommended to use that version (or above); all of these packages can be installed via pip3:

```
pip3 install bitstring cryptography pyzmq sympy
```

Other packages that are used (such as "sys" and "hashlib") are part of the standard Python library and are already available with each Python3 installation.

Another tool used is the "GNU Make" software (tested version 3.81). While not strictly required to run the scripts, it simplifies the use of this implementation. "Make" can be downloaded at the *official GNU software page*.

5.2 Running the script

In order to run the script, the user must open two different terminals in the project root directory: these two terminals will enact the two different parties. If "GNU Make" is installed, it is sufficient to execute `make alice` in order to run Alice or `make Bob` to run Bob; otherwise, it is also possible to run the two parties by executing `python3 src/mpc.py` ***`party`***, where ***`party`*** is either "alice" or "bob". This latter method also allows to change the output verbosity with the option `-l` ***`logging_value`***, where ***`logging_value`*** can be instantiated with "debug", "info", "warning", "error", "critical", adding some additional information on the OT protocol inputs.

After running the script, both of Alice's and Bob's terminals will request some input from the user. Valid inputs are positive and negative integers, while all other types of inputs (for example floating point or strings) will be discarded. After entering both of Alice's and Bob's inputs the computation will start and the result will be printed on both terminals.

When the computation has ended the result will be compared to the actual expected result computed in a classical way: if it is correct, the program will print a message and exit, while if it is incorrect, it will print an error message along with the expected result and the one obtained from the circuit.

5.2.1 Running example (Alice)

```
$: make alice
python3 src/mpc.py alice
Enter Alice's inputs: 2 4 6
Enstablishing communication with Bob...
Communication enstablished
Sent circuit to Bob
Input (A):  00000000000000000000000000000110
Received binary result:  00000000000000000000000000000110
Multiplexer bit: 0
Result converted to decimal: 6

Verifying correctness of the computation...
Max computed successfully:  6
```

5.2.2 Running example (Bob)

```
$: make bob
python3 src/mpc.py bob
Enter Bob's inputs: 1 3 5
Waiting for Alice...
Enstablished communication with Alice
Received circuit from Alice
Input (B):  00000000000000000000000000000101
Circuit input (-B): 11111111111111111111111111111011
Binary result of the circuit:  00000000000000000000000000000110
Multiplexer bit: 0
Result converted to decimal: 6

Verifying correctness of the computation...
Max computed successfully:  6
```

References

[1] O. Biçer. Efficiency Optimizations on Yao's Garbled Circuits and Their Practical Applications. *ArXiv*, abs/1703.03473, 2017.

[2] Andrew Chi-Chih Yao. How to generate and exchange secrets. *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986.