Project documentation

Denis D'Ambrosi

May 24, 2023

1 Project overview

This technical report presents a practical implementation of Yao's protocol, a two-party secure computation protocol that allows parties to jointly compute functions on private inputs without disclosing any information about those inputs to each other. Firstly introduced in 1986 by Andrew Yao [Yao86], this implementation utilizes garbled circuits and oblivious transfers to obtain secure computation between two parties. Within this paper we will describe some aspects of the design and the actual realization - including cryptographic primitives utilized, garbled circuit construction, and communication protocol - and will provide the documentation for running the example.

2 The circuit

By default, the script will simulate the execution of an 8-bit adder [1] with an additional component that prevents the risk of overflow. Before proceding with the explaination of the algorithm, it would be a good idea to explain a little futher how the base logical architecture works. An 8-bit logical adder is a digital circuit that can add two 8-bit binary numbers together. It consists of a half-adder and seven full adder subcircuits connected in cascading fashion:

- A half adder is a logical circuit that performs the addition of two single-bit binary numbers. The circuit consists of an XOR gate, that produces the equivalent sum bit, and an AND gate, which computes the carry bit. A diagram of such component is illustrated in figure 1

- On the other hand, each full-adder subcircuit requires three input bits: two bits to be added and a carry-in bit from the previous stage. After processing these inputs, it produces a sum bit and propagates that carry-out bit onto the carry-in input of the next stage. The final carry-out bit is discarded, and the sum bits produced by each full-adder subcircuit combined to produce the final result. A diagram that depicts a full-adder is shown in figure 2.

[1] Note that the number of bits of the representation can be changed as a command-line argument

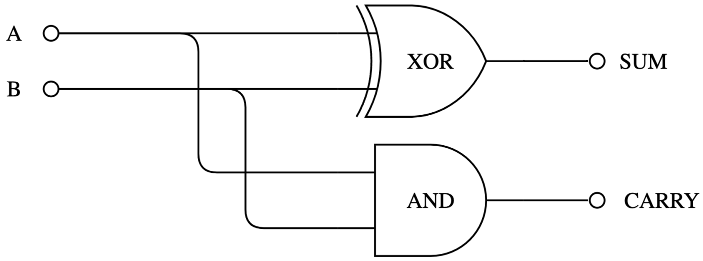

Figure 1: Half adder diagram

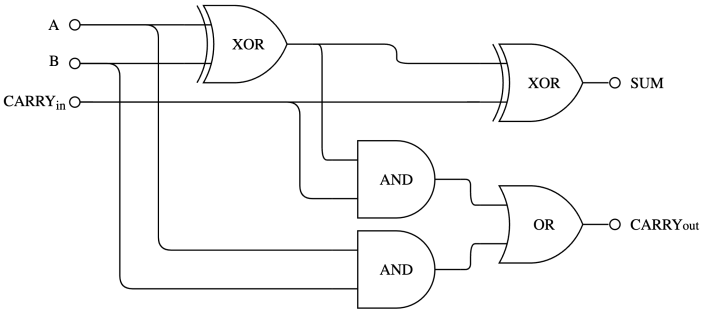

Figure 2: Full adder diagram

A general visual overview of a complete 8-bit adder that uses a half-adder and 7 full-adders as subcomponents can be seen in figure 3

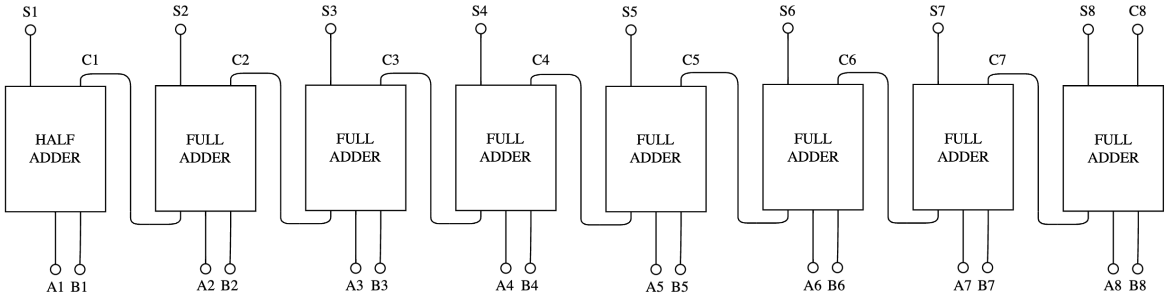

Figure 3: 8-bit adder diagram

An n-bit adder is generally used to sum two n-bit values into an n-bit number. In our case, since (as we will see in the following sections) the two parties cannot know the other partecipant's value, the computation could possibly lead to overflow. A simple trick to avoid this issue is to take into consideration both the last sum bit and the most significant input bits (thus bits $A8$, $B8$ and $S8$ in the above figure 3) and to use this information to extend the representation of the result into an $n+1$-bit value. The specification of this project requires the implementation to work with integers (positive and negative numbers alike), thus we need to ensure that the last bit does

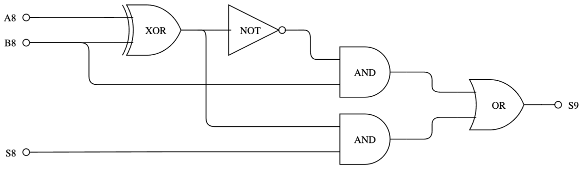

Figure 4: Diagram of the if-else circuit for the 9^{th} bit after an 8-bit adder.

not accidentally flip the sign when dealing with 2's complement arithmetic. In order to do so, all we have to do is to check the if the last two input bits match: if they match and are both equal to 1, we are summing two negative numbers, thus $S9$ will be 1. Similarly, if $S8$ is equal to 1 and the input bits are different, we are dealing again with a negative number, so $S9$ must be 1. On the other hand, in the other cases, $S9$ will be set to 0 since we are clearly dealing with positive numbers. The circuit used to compute this check is summarized in figure 4.

3 Yao's protocol

Alice's role in Yao's protocol [Yao86] is to garble the circuit that computes the joint function $f(x, y)$ using their private inputs x and y, then send it over a secure communication channel to Bob. Garbling the circuit requires assigning random labels to both the input wires of her input x and the output wires, then encrypting each gate's truth table using these labels as keys. Bob takes Alice's garbled circuit and evaluates it using both of his own inputs and Alice's, then sends the output in its current state back to Alice.

Alice's main responsibility is making sure that the circuit remains secure, without disclosing any private information about her private input x to Bob.

Bob's main responsibility is to guarantee the proper evaluation of the garbled circuit without disclosing any private input y to Alice. To accomplish this, Bob utilizes the same garbling scheme and secure communication protocol as Alice to guarantee that the inputs are kept private.

The actual algorithms executed by the two parties are described below:

```
Function Alice($x$):
    truth_table ← derive_truth_table(circuit);
    keys ← generate_encryption_keys();
    encrypted_circuit ← encrypt_rows(truth_table, keys);
    garbled_circuit ← permutate_rows(encrypted_circuit);
    $x$_labels ← [];
    for $i \in \{1..n\}$ do
        $x$_labels[i] ← obtain_label(circuit, Alice, i, $x$[i], keys);
    end
    send(Bob, (garbled_circuit, $x$_labels));
    $y$_labels_0 ← []; $y$_labels_1 ← [];
    for $i \in \{1..n\}$ do
        $y$_labels_0[i] ← obtain_label(circuit, Bob, i, 0, keys);
        $y$_labels_1[i] ← obtain_label(circuit, Bob, i, 1, keys);
    end
    for $i \in \{1..n\}$ do
        send_oblivious_transfer(Bob, $y$_labels_0[i], $y$_labels_1[i]);
    end
    result ← receive(Bob);
End Function
```

Algorithm 1: Alice's steps

```
Function Bob($y$):
    garbled_circuit, $x$_labels ← receive(Alice);
    $y$_labels = [];
    for $i \in \{1..n\}$ do
        $y$_labels ← receive_oblivious_transfer(Alice, y[i])
    end
    result ← evaluate_circuit(garbled_circuit, $x$_labels, $y$_labels);
    send(Alice, result);
End Function
```

Algorithm 2: Bob's steps

Note that in this pseudocode we assume 3 things:

1. Both parties already know the circuit, so Alice knows what to garble and Bob knows how to evaluate it after receiving the keys;

2. We have at our disposal a black-box function `obtain_label(circuit, party, i, v, keys)` that selects the key (belonging to `keys`) corresponding to the `i`th input wire of `party` in the shared encoding of `circuit` if it had value `v`. In practice this is done through a dictionary lookup;

3. We have at our disposal an oblivious transfer procedure and the relative primitives `send_oblivious_transfer(receiver, message1, message2)` and `receive_oblivious_transfer(sender, message_number)`.

4 Project structure

My implementation for this project is heavily inspired by the `garbled-circuit` [RR22] public library. In the following list, I will summarize the features of each file, but, for the sake of briefness, I will focus in particular only on the functions added to the repository.

- `main.py`: this file actually represents the "entry point" to our program: it collects the command line arguments provided by the user and creates the correct objects for the execution of the protocol according to the options provided as input. After the istantiation, it also starts the exchange and (if run as Alice) checks the results.

- `parties.py`: this module contains the implementation of the algorithms described in the previous section: following the class hierarchy defined in the `main.py` file of the original repository, it provides the classes needed for both Alice's and Bob's procedures. According to the role we want to impersonate, we have to instantiate and call these objects in a slighty different way:

  - `Alice` is initialized by passing the paths to a JSON file containing the encoding of a circuit according to the `garbled-circuit` repository, the input file and the output file. After initialisation, this party can begin the exchange (assuming we have already created correctly an instance of `Bob` listening on the same port) by calling its method `start()`.
  - `Bob` is initialised by providing the path to an input file. After creation, it can check for incoming connections through its `listen()` method.

  Note that the constructors of both of these classes have an additional (optional) argument, `oblivious_transfer`, which defines whenever we want to use the homonimous protocol during the exchange; this parameter is set to `True` by default.

- `yao.py`: this module contains all the logic needed for the actual Yao's protocol realization (the code for used for actual cryptography, garbling tables, etc...); this file is a virtually untouched copy of the the original module defined in the aforementioned repository: I only modified the encryption/decryption functions so that instead of calling `cryptography` [Aut23] `Fernet` black-box procedures they directly implement *AES*, following this project's implementation requirements.

- `ot.py`: this module implements the logic needed for the oblivious transfer procedure. No changes were done to the original file of the `garbled-circuit` library.

- `util.py`: this module provides miscellaneous additional functionalities to the rest of the program to streamline the designing procedure of the more complex functions. Most of the code available within this file is directly taken from the corrispondent module of the aforementioned public repository. In addition, I included the following procedures:

- read_input_data, that takes as input the path to an input file and reads and aggregates its content to obtain the actual value to compute the protocol on. Although my implementation computes the sum of the elements of two sets, I made this function modular so that the aggregation procedure could be easily swapped for another set-to-integer function (for example a maximum or a minimum).
- convert_to_binary_list, which takes the integer value computed through the previous function and maps it into the corresponding circuit input in two's complement. Note that also this function is parametrized on the number of bits used for the representation, so anyone could theoretically change the size of the circuit and of the binary representation pretty easily by modifying the default values.
- convert_to_decimal, that decodes the circuit's output into an integer result.
- save_results, which dumps the value computed through the previous function into an output file.
- verify, that computes locally the sum of the aggregated values (by reading the corresponding input files) and checks that result is coherent with the content of the output file.
- generate_circuit, which allows for the procedural generation of the circuit according to the library's JSON specification. Again, also this function is parametrized on the size of the input, but it is invoked with a default value of 8 bits.
- generate_and_save_circuit, that saves locally a copy of the above generated circuit.

5 Usage

To run the program (which was written in Python version 3.11.2), you have to execute the following 4 steps:

1. Install the required packages listed in file requirements.txt (for example with the command `pip install -r requirements.txt`);
2. Write a set of whitespace-separated integers in the file input_alice.txt;
3. Write a set of whitespace-separated integers in the file input_bob.txt;
4. Run the script as Bob by executing `python3 main.py bob`;
5. In another terminal, run the script as Alice by executing `python3 main.py alice`.

The protocol will be quickly run and you will be able to check the correctness of the execution in Alice's terminal since it will output a success/failure message. If, however, you want to manually check for the correctness of the protocol, you can look at the content of the file `output.txt`, where you will find the sum of the aggregated values of the two parties.

My implementation includes some additional command-line options for additional flexibility:

- When we run the program as Alice, we can specify the path of the file to save the circuit to through the `--circuit` flag (example: `python3 main.py alice --circuit circuitfile.json`);

- When we execute the program as Alice, we can inhibit the use of oblivious transfers in the protocol through the `--no-oblivious-transfer` flag (example: `python3 main.py alice --no-oblivious-transfer`). Note that, from a purely functional point of view, this doesn't change the behaviour of the system.

- When we run the program as either of the two parties, we can specify the input file for the data through the `--[party]` flag (example: `python3 main.py bob --bob bobfile.txt`).

- The `--bits` flag allows us to specify the size of the circuit's input and the data representation (example: `python3 main.py bob --bits 12`). Note that this option **must** be mirrored on both terminals.

- Finally, we can se the logging level through the `--loglevel` flag (example: `python3 main.py alice -loglevel info`).

Note also that the program will raise an Exception if the sum of the numbers provided in the input file exceeds the limits of their binary representation. In particular, since the exchange works with signed integers, the input values for an n-size circuit must belong to the interval $[-2^{n-1}, 2^{n-1} - 1]$.

References

[Aut23] Python Cryptography Authority. cryptography. `https://github.com/pyca/cryptography`, 2023.

[RR22] Oliver Roques and Emmanuelle Risson. garbled-circuit. `https://github.com/ojroques/garbled-circuit`, 2022.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE Computer Society Press, 1986.