Integer Set Secure Computation through Yao's Protocol - Documentation

Luca Campa

January 10, 2022

1 Introduction

This project implements a two-party secure function evaluation using Yao's garbled circuit protocol. The original implementation is taken from the following GitHub Repository: https://github.com/ojroques/garbled-circuit.
In this model, Alice and Bob compute a function on their inputs without sharing the value of their inputs with the opposing party. Alice is the circuit creator (the garbler) while Bob is the circuit evaluator. Alice creates the yao circuit and sends it to Bob along with her encrypted inputs. Bob then computes the results and sends them back to Alice [1]. The inputs from Alice and Bob will consist of sets of integers. The architecture supports different sets' cardinalities and integers of any size. The two supported functions are:

1 The sum of the values of Alice and Bob sets.

2 The common values between Alice and Bob sets.

The circuit used by Yao's protocol in order to compute the function will be dinamically build at each Alice connection to Bob. It will depend on the function we want to compute and, in particular:

- the sum will depend on the maximum length of the binary representation we are going to deal with.
- the common values (set comparison) will depend on the Alice's set cardinality and on the maximum length of the binary representation we are going to deal with.

[1]https://github.com/ojroques/garbled-circuit/blob/master/README.md

2 Installation

Code is written for Python 3.*. The dependencies are:

- **ZeroMQ** for communications
- **AES** for encryption of garbled tables (mode CBC) - from cryptography
- **SymPy** for prime number manipulation
- **atexit** in order to intercept the keyboard combination Ctrl-C , already with python3.*
- **pickle** already with python3.*

To install all dependencies:

```
$ pip3 install --user pyzmq cryptography sympy
```

3 Usage

The path of the files I will go to use are relative to the folder **/src**. So, when I say main.py I mean I'm within the folder *src*. If you want to run the code from outside this folder, pay attention to write the correct path to main.py (e.g. **/src/main.py**).

3.1 Communication

1 All tests are done on the local network. You can edit the network information in **util.py**.

2 Run the server (Bob): `$ python main.py bob`.

3 In another terminal, run the client (Alice): `$ python main.py <--operation=> <--mode=> alice`. The default operation is the sum of the sets' values. The default mode is circuit (inputs and outputs).

Alice will print her inputs alongside the corresponding outputs from the evaluator Bob. Alice does not know Bob's inputs, then Bob's inputs will be printed by Bob himself.

3.2 Alice's usage

To print Alice's inputs and the computed outputs of a circuit:

```
$ python main.py alice -m circuit
```

or

```
$ python main.py alice
```

To print a clear representation of the garbled tables of a circuit:

```
$ python main.py alice -m table
```

To change the operation (0 = sum, 1 = common values between two sets)

```
$ python main.py alice -o 1
```

Parameters summary:

- –operation or -o in order to change the operation to be computed (0 : sum, 1: common values).
- –mode or -m the printing mode for tests (circuit, table)

3.3 Bob's usage

To run the server (Bob):

```
$ python main.py bob
```

Follow the requests and type in Bob's set of integers. In order to change Bob's set without restarting the entire process, you can press the termination sequence **Ctrl-C**. This will show you a simulated shell where you can use 4 possible commands:

- *exit* in order to close the process
- *help* in order to show what are the available commands and their usage
- *new set* in order to assign a new integer set to Bob
- *continue* in order to listen again to Alice's messages and evaluate them.

If you want to listen again to Alice's messages (without changing the set or after having changed the set) you need to use the command: **continue**.

4 Architecture

The project is composed of 5 python files:

- **main.py** implements Alice side, Bob side and local tests.
- **yao.py** implements:
  - Encryption and decryption functions.
  - Evaluation function used by Bob to get the results of a yao circuit
  - GarbledCircuit class which generates the keys, p-bits and garbled gates of the circuit.

    - GarbledGate class which generates the garbled table of a gate.

- **ot.py** implements the oblivious transfer protocol.
- **util.py** implements many functions related to network communications and asymmetric key generation.[2]
- **circuit_generator.py** implements the necessary functions used to create the circuit for Yao's protocol. They create a json format of the circuit by following the structure suggested by the original project documentation. The created json circuits will be saved to json files under the directory circuits.

5 JSON Circuit

A function is represented as a boolean circuit using some of the available gates from the original project:

- NOT (1-input gate)
- AND
- OR
- XOR

A few assumptions are made (from the original documentation):

- Bob knows the boolean representation of the function. Thus the principle of "No security through obscurity" is respected.
- All gates have one or two inputs and only one output.
- The outputs of lower numbered gates will always be wired to higher numbered gates and/or be defined as circuit outputs.
- The gate id is the id of the gate's output.

An example of the circuit representation can be found within the original documentation.

[2] https://github.com/ojroques/garbled-circuit/blob/master/README.md

6 Outputs Examples

Here there is an example of the Bob's output and its interpretation:

```
$ python3.8 main.py bob
Enter the number of integers of Bob's set: 5
Enter the list items separated by space: 1 2 3 4 5
Alice asks for max bit length ...
Received set_sum
Bob[6, 7, 8, 9, 10] = 1 1 1 1 0
Alice asks for max bit length ...
Received set_cmp
Bob[19, 20, 21, 22, 23, 24] = 1 0 0 0 0 0
Bob[19, 20, 21, 22, 23, 24] = 0 1 0 0 0 0
Bob[19, 20, 21, 22, 23, 24] = 1 1 0 0 0 0
Bob[19, 20, 21, 22, 23, 24] = 0 0 1 0 0 0
Bob[19, 20, 21, 22, 23, 24] = 1 0 1 0 0 0
```

Interpretation: Alice does not know Bob's inputs, then she can't print them. Bob's inputs will be printed by Bob himself. The message *Alice asks for max bit length* means that Alice needed to know the number of bits (at maximum) in order to build the correct circuit. The message *Received <name_of_the_circuit>* tells you what operation they are going to compute together.

Here there is an example of the 'circuit' Alice's output and its interpretation (operation = SUM):

```
$ python3.8 main.py -o 0 alice
Enter the number of integers of Alice's set: 3
Enter the list items separated by space: 1 2 3
The sum of the elements from Bob and Alice should be: 21
======== set_sum ========
Alice[1, 2, 3, 4, 5] = 0 1 1 0 0
Outputs[11, 15, 20, 25, 30, 32] = 1 0 1 0 1 0
The sum of the elements is: 21
[CORRECT] The yao's output is equal to the one computed in normal way.
```

Interpretation: as you can see from the operation parameter, the chosen operation is SUM. Alice prints her inputs alongside the outputs computed by using Yao's protocol. The last line is a check for the correctness of the result compared to the same operation computed in normal way.

Here there is an example of the 'circuit' Alice's output and its interpretation (operation = COMMON VALUES):

```
$ python3.8 main.py -o 1 alice
Enter the number of integers of Alice's set: 3
Enter the list items separated by space: 1 4 23
The common elements between the Bob and Alice's sets are: {1, 4}
```

```
======== set_cmp ========
Alice[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18] =
    1 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 1 0
Outputs[77, 82, 87, 92, 97, 102, 107] = 1 1 0 0 0 0 0
Outputs[77, 82, 87, 92, 97, 102, 107] = 0 0 0 0 0 0 0
Outputs[77, 82, 87, 92, 97, 102, 107] = 0 0 0 0 0 0 0
Outputs[77, 82, 87, 92, 97, 102, 107] = 1 0 0 1 0 0 0
Outputs[77, 82, 87, 92, 97, 102, 107] = 0 0 0 0 0 0 0
Common values:
1 4 [CORRECT] The yao's output is equal to the one computed in normal
   way.
```

Interpretation: as you can see from the operation parameter, the chosen operation is COMMON VALUES - (set compare). Alice prints her inputs. Moreover she prints the outputs received from Bob. Each output has to be interpreted in the following way:

- the output is composed by two parts: the first bit has the meaning of equality (0: not equal, 1: equal); the remaining bits are the binary representation of the common value (inverse order).

The last line is a check for the correctness of the result compared to the same operation computed in normal way.

Here there is an example of the tables printed by Alice by using the parameter –mode=table.

```
$ python3.8 main.py -o 0 -m table alice
Enter the number of integers of Alice's set: 3
Enter the list items separated by space: 1 2 3
The sum of the elements from Bob and Alice should be: 21
======== set_sum ========
P-BITS: {1: 0, 2: 1, 3: 1, 4: 1, 5: 1, 6: 0, 7: 1, 8: 1, 9: 0, 10: 0,
   11: 0, 12: 0, 13: 1, 14: 0, 15: 0, 16: 0, 17: 1, 18: 0, 19: 0, 20:
   1
GATE: 11, TYPE: XOR
[0, 0]: [1, 0][6, 0]([11, 0], 0)
[0, 1]: [1, 0][6, 1]([11, 1], 1)
[1, 0]: [1, 1][6, 0]([11, 1], 1)
[1, 1]: [1, 1][6, 1]([11, 0], 0)
GATE: 12, TYPE: AND
[0, 0]: [1, 0][6, 0]([12, 0], 0)
[0, 1]: [1, 0][6, 1]([12, 0], 0)
[1, 0]: [1, 1][6, 0]([12, 0], 0)
[1, 1]: [1, 1][6, 1]([12, 1], 1)
GATE: 13, TYPE: XOR
[0, 0]: [2, 1][7, 1]([13, 0], 1)
[0, 1]: [2, 1][7, 0]([13, 1], 0)
[1, 0]: [2, 0][7, 1]([13, 1], 0)
[1, 1]: [2, 0][7, 0]([13, 0], 1)
GATE: 14, TYPE: AND[0, 0]: [2, 1][7, 1]([14, 1], 1)
```

```
[0, 1]: [2, 1][7, 0]([14, 0], 0)
[1, 0]: [2, 0][7, 1]([14, 0], 0)
[1, 1]: [2, 0][7, 0]([14, 0], 0)
GATE: 15, TYPE: XOR
[0, 0]: [13, 1][12, 0]([15, 1], 1)
[0, 1]: [13, 1][12, 1]([15, 0], 0)
[1, 0]: [13, 0][12, 0]([15, 0], 0)
[1, 1]: [13, 0][12, 1]([15, 1], 1)
GATE: 16, TYPE: AND
[0, 0]: [13, 1][12, 0]([16, 0], 0)
[0, 1]: [13, 1][12, 1]([16, 1], 1)
[1, 0]: [13, 0][12, 0]([16, 0], 0)
[1, 1]: [13, 0][12, 1]([16, 0], 0)
GATE: 17, TYPE: OR
[0, 0]: [14, 0][16, 0]([17, 0], 1)
[0, 1]: [14, 0][16, 1]([17, 1], 0)
[1, 0]: [14, 1][16, 0]([17, 1], 0)
[1, 1]: [14, 1][16, 1]([17, 1], 0)
GATE: 18, TYPE: XOR
[0, 0]: [3, 1][8, 1]([18, 0], 0)
[0, 1]: [3, 1][8, 0]([18, 1], 1)
[1, 0]: [3, 0][8, 1]([18, 1], 1)
[1, 1]: [3, 0][8, 0]([18, 0], 0)
GATE: 19, TYPE: AND
[0, 0]: [3, 1][8, 1]([19, 1], 1)
[0, 1]: [3, 1][8, 0]([19, 0], 0)
[1, 0]: [3, 0][8, 1]([19, 0], 0)
[1, 1]: [3, 0][8, 0]([19, 0], 0)
GATE: 20, TYPE: XOR
[0, 0]: [18, 0][17, 1]([20, 1], 0)
[0, 1]: [18, 0][17, 0]([20, 0], 1)
[1, 0]: [18, 1][17, 1]([20, 0], 1)
[1, 1]: [18, 1][17, 0]([20, 1], 0)
GATE: 21, TYPE: AND
[0, 0]: [18, 0][17, 1]([21, 0], 0)
[0, 1]: [18, 0][17, 0]([21, 0], 0)
[1, 0]: [18, 1][17, 1]([21, 1], 1)
[1, 1]: [18, 1][17, 0]([21, 0], 0)
GATE: 22, TYPE: OR
[0, 0]: [19, 0][21, 0]([22, 0], 0)
[0, 1]: [19, 0][21, 1]([22, 1], 1)
[1, 0]: [19, 1][21, 0]([22, 1], 1)
[1, 1]: [19, 1][21, 1]([22, 1], 1)
GATE: 23, TYPE: XOR
[0, 0]: [4, 1][9, 0]([23, 1], 0)
[0, 1]: [4, 1][9, 1]([23, 0], 1)
[1, 0]: [4, 0][9, 0]([23, 0], 1)
[1, 1]: [4, 0][9, 1]([23, 1], 0)
GATE: 24, TYPE: AND
[0, 0]: [4, 1][9, 0]([24, 0], 1)
[0, 1]: [4, 1][9, 1]([24, 1], 0)
[1, 0]: [4, 0][9, 0]([24, 0], 1)
```

```
[1, 1]: [4, 0][9, 1]([24, 0], 1)
GATE: 25, TYPE: XOR
[0, 0]: [23, 1][22, 0]([25, 1], 1)
[0, 1]: [23, 1][22, 1]([25, 0], 0)
[1, 0]: [23, 0][22, 0]([25, 0], 0)
[1, 1]: [23, 0][22, 1]([25, 1], 1)
GATE: 26, TYPE: AND
[0, 0]: [23, 1][22, 0]([26, 0], 1)
[0, 1]: [23, 1][22, 1]([26, 1], 0)
[1, 0]: [23, 0][22, 0]([26, 0], 1)
[1, 1]: [23, 0][22, 1]([26, 0], 1)
GATE: 27, TYPE: OR
[0, 0]: [24, 1][26, 1]([27, 1], 1)
[0, 1]: [24, 1][26, 0]([27, 1], 1)
[1, 0]: [24, 0][26, 1]([27, 1], 1)
[1, 1]: [24, 0][26, 0]([27, 0], 0)
GATE: 28, TYPE: XOR
[0, 0]: [5, 1][10, 0]([28, 1], 1)
[0, 1]: [5, 1][10, 1]([28, 0], 0)
[1, 0]: [5, 0][10, 0]([28, 0], 0)
[1, 1]: [5, 0][10, 1]([28, 1], 1)GATE: 29, TYPE: AND
[0, 0]: [5, 1][10, 0]([29, 0], 0)
[0, 1]: [5, 1][10, 1]([29, 1], 1)
[1, 0]: [5, 0][10, 0]([29, 0], 0)
[1, 1]: [5, 0][10, 1]([29, 0], 0)
GATE: 30, TYPE: XOR
[0, 0]: [28, 0][27, 0]([30, 0], 1)
[0, 1]: [28, 0][27, 1]([30, 1], 0)
[1, 0]: [28, 1][27, 0]([30, 1], 0)
[1, 1]: [28, 1][27, 1]([30, 0], 1)
GATE: 31, TYPE: AND
[0, 0]: [28, 0][27, 0]([31, 0], 0)
[0, 1]: [28, 0][27, 1]([31, 0], 0)
[1, 0]: [28, 1][27, 0]([31, 0], 0)
[1, 1]: [28, 1][27, 1]([31, 1], 1)
GATE: 32, TYPE: OR
[0, 0]: [29, 0][31, 0]([32, 0], 1)
[0, 1]: [29, 0][31, 1]([32, 1], 0)
[1, 0]: [29, 1][31, 0]([32, 1], 0)
[1, 1]: [29, 1][31, 1]([32, 1], 0)
```

Interpretation: as you can see from the mode parameter, the chosen print mode is *table*. Alice prints the garbled table for each gate within the circuit.

7 Implementation Choices

Bit Length In order to deal with variable integers' length, two solutions can be found:

- using a fixed maximum size (e.g. 32 bit)

- Alice will communicate with the other party in order to understand what is the maximum number of bits required to compute the result. In this way there will be less communication cost compared to the previous solution. (MY CHOICE)

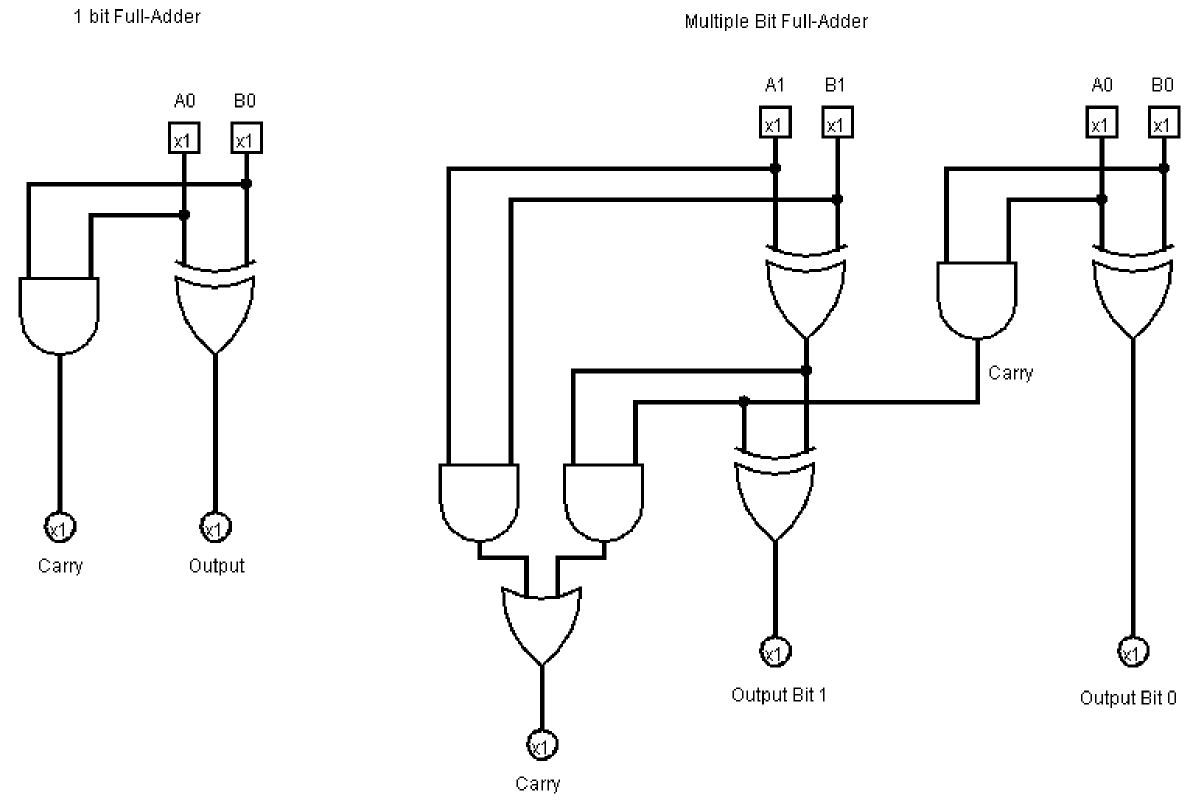

Figure 1: Sum Circuit

Set Sum Both Alice and Bob choose a set of integers. By the fact that they want to compute the sum of all these values within the sets, they precompute the sum of their respective set's values. By doing so, they will not reveal to the other party any additional information such as the number of values within their sets and the value of these integers. The circuit for the sum is generated by the *create_circuit* method within **circuit_generator.py**. It will be built by combining multiple full-adders as you can see from the Figure 1.

Common Values (set compare) Both Alice and Bob choose a set of integers. Alice will build the circuit on the base of her set's cardinality. The circuit will receive all Alice's values and one value at each time from Bob. The value from Bob will be iteratively compared with all the Alice's inputs. The circuit will return a bit meaning the equality with one of the Alice's values alongside the common value itself. The

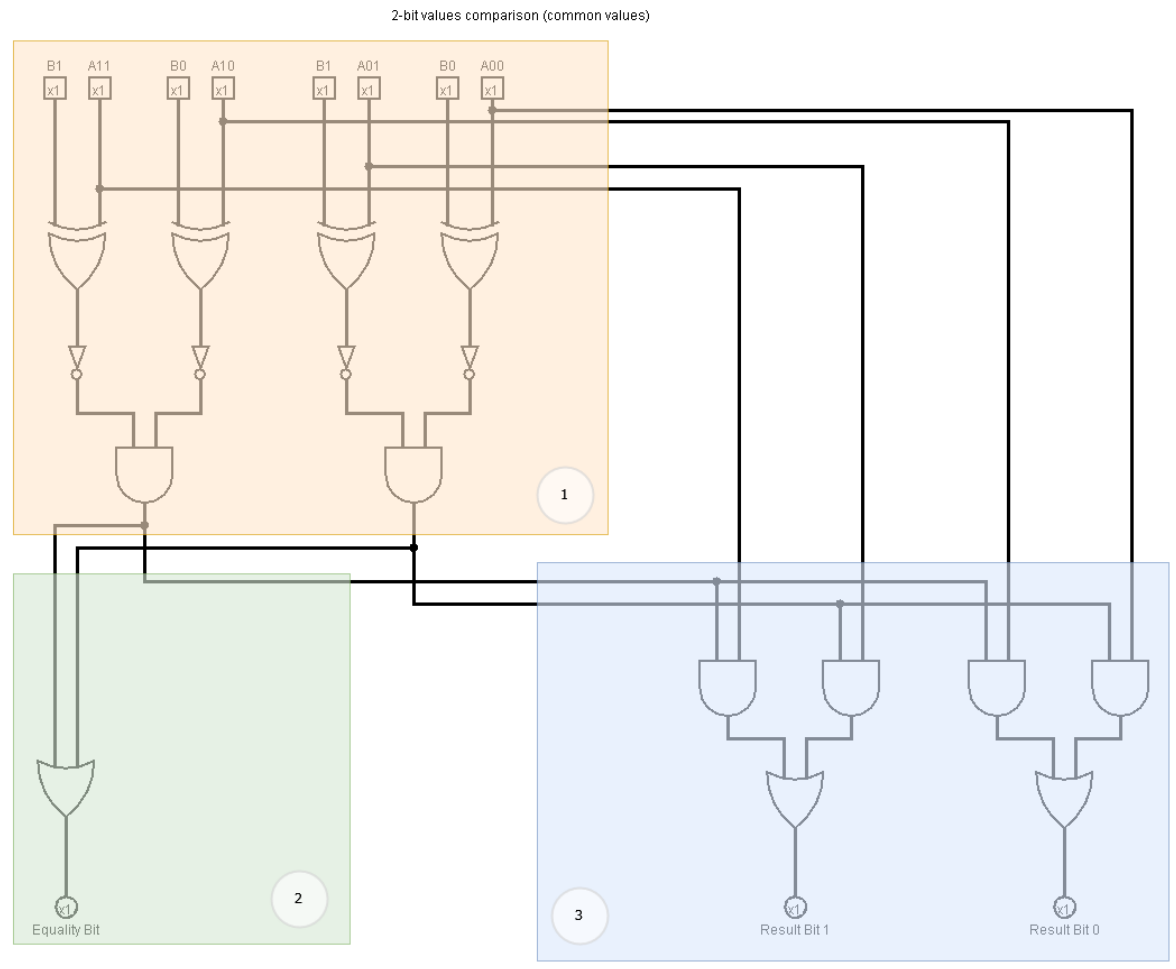

Figure 2: Common Value Circuit

circuit for the comparison of the sets in order to get the common value is composed by three sections combined together (Figure 2):

1 The loop for the comparison of Bob's input with each Alice's value. It returns an equality bit for each comparison.

2 The resulting equality bit from the previous ones.

3 By using the equality bits from the point (1) it returns the common value itself if exists (otherwise the bits are set to 0).

Bob will permutate his set values before sending them for the comparison.

Yao's encryption As for the encryption system used by these Yao's implementation, I chose AES mode CBC and the IV will be prepended to the ciphertext from the encryption function. An other approach could be to use the same IV for each computation but it would not be secure.

8 Pseudo-Code

Set Sum Algorithm 1 shows how the circuit for the sum of two values will be dinamically created on the base of the bit length it has to deal with.

```
Data: B: bit_length, s: start index
Result: Components list of the JSON circuit
index = s;
step = B;
gates = [ ];
alice = [ ];
bob = [ ];
outputs = [ ];
carry_index = 0;
for $i \leftarrow 0$ to $B$ by 1 do
    alice.insert(index);
    index = index + 1;
end
for $i \leftarrow 0$ to $B$ by 1 do
    bob.insert(index);
    index = index + 1;
end
for $i \leftarrow 1$ to $B + 1$ by 1 do
    if $i$ == 1 then
        gates.insert("id": index, "type": "XOR", "in": [i,i+step]);
        outputs.insert(index);
        index = index + 1;
        gates.insert("id": index, "type": "AND", "in": [i,i+step]);
        carry_index = index;
        if $B == i$ then
            outputs.insert(carry_index);
        end
        index = index + 1;
    else
        gates.insert("id": index, "type": "XOR", "in": [i,i+step]);
        xor_result = index;
        index = index + 1;
        gates.insert("id": index, "type": "AND", "in": [i,i+step]);
        and_result_1 = index;
        index = index + 1;
        gates.insert("id": index, "type": "XOR", "in": [xor_result,carry_index]);
        outputs.insert(index);
        index = index + 1;
        gates.insert("id": index, "type": "AND", "in": [xor_result,carry_index]);
        and_result_2 = index;
        index = index + 1;
        gates.insert("id": index, "type": "OR", "in": [and_result_1, and_result_2]);
        carry_index = index;
        index = index + 1;
        if $B$ == 1 then
            outputs.insert(carry_index);
        end
    end
end
return alice, bob, outputs, gates;
```

Algorithm 1: How to build the circuit for sum of two values

Common Values Algorithm 2 shows how to build the circuit for the comparison of one Bob's value with multiple values from Alice's set. It will be dinamically created depending on the max bit length of the values and on the cardinality of the Alice's set.

```
Data: B: bit_length, C: alice's set cardinality
Result: Components list of the JSON circuit
gates = [ ];
alice = [ ];
bob = [ ];
outputs = [ ];
index = 0;
for $i \leftarrow 0$ to $B * C$ by 1 do
    index = index + 1;
    alice.insert(index);
end
for $i \leftarrow 0$ to $B$ by 1 do
    index = index + 1;
    bob.insert(index);
end
index = index + 1;
equality_gates_indexes = [ ];
for $value_index$ \leftarrow 0 to $C$ by 1 do
    not_indexes = [ ];
    for $bit_index$ \leftarrow 0 to $B$ by 1 do
        xor_index = index;
        gates.insert("id": index, "type": "XOR", "in": [alice[b*value_index+bit_index], bob[bit_index]]);
        index = index + 1;
        not_index = index;
        gates.insert("id": index, "type": "NOT", "in": [xor_index]);
        not_indexes.insert(not_index);
        index = index + 1;
    end
    and_index = not_indexes[0];
    for $i \leftarrow 1$ to $length(not_indexes)$ by 1 do
        gates.insert("id": index, "type": "AND", "in": [and_index, not_indexes[i]]);
        and_index = index;
        index = index + 1;
    end
    equality_gates_indexes.insert(and_index);
end
equality_index = equality_gates_indexes[0];
for $i \leftarrow 1$ to $length(equality_gates_indexes)$ by 1 do
    gates.insert("id": index, "type": "OR", "in": [equality_index, equality_gates_indexes[i]]);
    equality_index = index;
    index = index + 1;
end
outputs.insert(equality_index)
for $bit_index \leftarrow 0$ to $B$ by 1 do
    check_gates = [ ];
    for $value\_index \leftarrow 0$ to $C$ by 1 do
        gates.insert("id": index, "type": "AND", "in": [alice[value_index*b+bit_index],
          equality_gates_indexes[value_index]]);
        check_gates.insert(index);
        index = index + 1;
    end
    first_or_index = check_gates[0];
    for $index_check\_gate \leftarrow 1$ to $length(check\_gates)$ by 1 do
        gates.insert("id": index, "type": "OR", "in": [first_or_index, check_gates[index_check_gate]]);
        first_or_index = index;
        index = index + 1;
    end
    outputs.insert(first_or_index);
end
return alice, bob, outputs, gates;
```

Algorithm 2: How to build the circuit for the comparison of one value with multiple values of the other party

In order to better understand what the circuit will do, I also provide a code representation of its behaviour (Algorithm 3). Putting the things together, Bob will loop on his values sending each of them to the circuit and sending the result to Alice.

```
Data: alice_set, bob_value
Result: Equality Bit, Common Value
equality_bit = 0;
common_value = 0;
for $i \leftarrow 0$ to $length(alice\_set) - 1$ by 1 do
    if bob_value == alice_set[i] then
        equality_bit = 1;
        common_value = alice_set[i];
    end
end
return equality_bit, common_value;
```

Algorithm 3: The same behaviour of the circuit shown as pseudo-code

Alice and Bob behaviours The following pseudo-code algorithms show methods' names which are not the same on the project. They are used only to make the behaviour of the two parties much more clear. They are intended to interact with each other in this way:

1 Alice and Bob agree on the max bit length they are going to deal with. (This is done in order to create a circuit which fits well for the chosen values).

2 Alice shares the chosen operation with Bob who agrees on it.

3 Alice creates the circuit and the tables and sends them to Bob.

4 Alice chooses her inputs

5 Bob chooses his inputs

6 Both interact with each other by using Oblivious Transfer.

7 Bob computes the result and sends it to Alice.

8 Alice checks the correctness of the result by comparing it with the same operation computed in normal way.

```
Data: alice_set, chosen_function
Result: Create the circuit and receives the result from Bob
exchange_max_bit_length_with_Bob();
share_chosen_operation();
create_circuit();
print(expected_output);
send_circuit_to_Bob();
send_inputs_with_ot();
receive_result_from_Bob();
check_result_correctness();
```

Algorithm 4: Alice's steps

```
Data: bob_set
Result: Computes the result and sends it to Alice
exchange_max_bit_length_with_Alice();
agree_on_chosen_operation();
receive_circuit();
receive_Alice_inputs();
choose_Bob_inputs();
compute_result_by_using_ot();
send_result_to_Alice();
```

Algorithm 5: Bob's steps - Circuit Evaluator