Introduction to Cybersecurity
Project documentation

Lorenzo Cian
Matrikelnummer 12136151

January 19, 2022

Contents

1 Introduction

This document constitutes the documentation for the project of the Introduction to Cybersecurity course. I worked on it by myself, and chose to implement the set intersection function, supporting floating point values which can be represented using 32 bits.

2 Structure of the project

My implementation is based on the suggested Python implementation [RR20] of Yao's garbled circuits [Yao86]. My project is based on the files from that package, most of which I modified to fit the requirements of the project.

Specifically, I made the following modifications/additions to the files you can find in the [RR20] Github repository:

- (circuits/eq32.json) added the eq32 Boolean circuit, which outputs 1 if all of the corresponding bits in the two 32 bits values held by the two parties are equal and 0 otherwise.

  It is implemented using 32 XOR gates and 31 non-XOR gates, to take advantage of the free XOR optimization.

  Specifically, given two 32-bit inputs $A = a_1 a_2 \dots a_{32}$ and $B = b_1 b_2 \dots b_{32}$, the circuit first uses 32 XOR gates, the i-th of which has inputs a_i and b_i.
  The output of the i-th XOR gate will be equal to 1 if and only if $a_i \neq b_i$, which implies $A \neq B$. So, we want the circuit to output 1 if and only if all the outputs of the 32 XOR gates are equal to 0: feeding the outputs of the XOR gates to a 32-bit NOR gate would do just that, but given that we can only use binary gates we need to simulate a 32-bit NOR gate by using 16 2-bit NOR gates and a total of 15 2-bit AND gates to join their outputs into a single bit.

  For simplicity of representation, in Figure 1 I show the eq8 circuit, a smaller circuit which has the same structure and behaviour as the eq32 circuit;

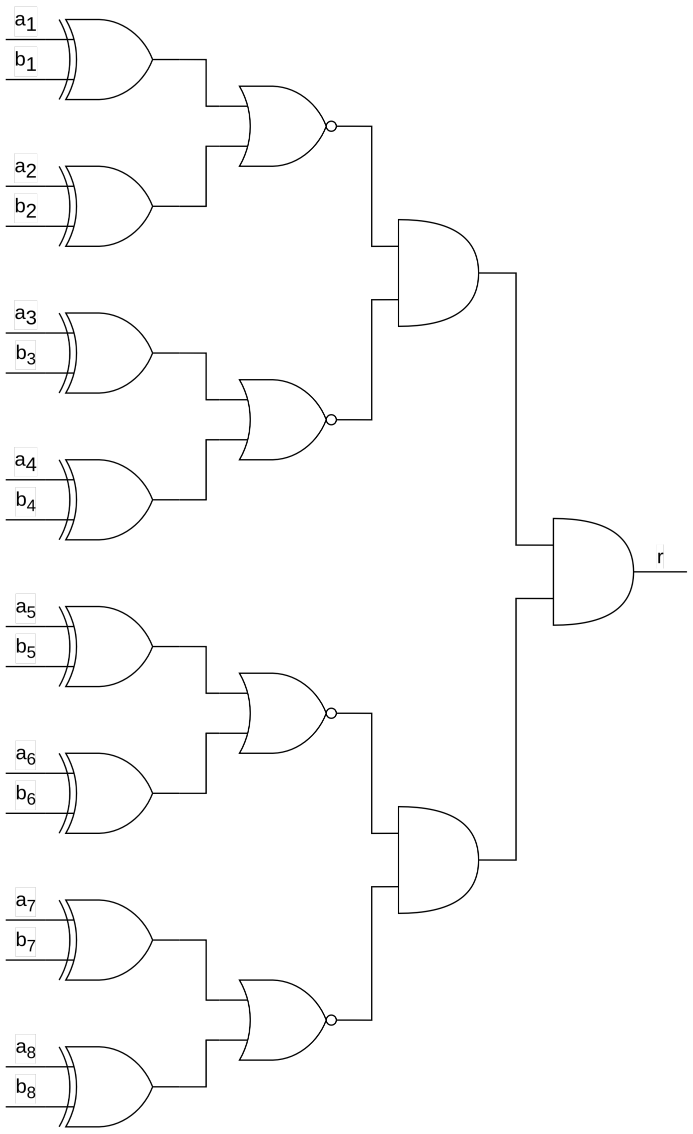

Figure 1: The eq8 circuit, which outputs 1 if and only if $a_i = b_i$ for $i = 1, \ldots, 8$. It exhibits the same structure and behaviour as the eq32 circuit but operates on 8-bit values.

- (psi.py) heavily modified the main.py script to implement the computation of the desired function, along with the other requirements of the project;
- (util.py) added the function parse_float_set to parse the input arguments, float_to_bit_list for the floating point conversion, along with a custom Logger class;
- (yao.py) modified yao.evaluate, yao.GarbledCircuit._gen_pbits, yao.GarbledCircuit._gen_keys to generate the keys, p-bits (a.k.a. signal bits), garbled tables and evaluate the circuit based on the free XOR optimization [KS08] of Yao's protocol.

3 How it works

The set intersection of two sets, each one held by one party, is computed privately by leveraging Yao's protocol [Yao86] and a set intersection protocol based on Pairwise-Comparisons [HEK12]. My implementation of the latter can be found in the methods of the classes Alice and Bob in the file psi.py and works as follows:

1. initially, Alice holds the set $A = \{a_0, \dots, a_{n-1}\}$; Bob holds the set $B = \{b_0, \dots, b_{m-1}\}$. Each party stores their set in a sorted list;
2. Alice sends a "PSI" message to which Bob responds, so that the two parties agree that they will compute the set intersection;
3. Bob sends the size of his set m to Alice;
4. for i in $0, \dots, n-1$ and j in $0, \dots, m-1$:
   Alice regenerates the eq32 circuit (the same circuit is used but the keys, the p-bits and consequently the tables are freshly generated, as reusing these elements would leading to losing the security properties of Yao's protocol),
   Alice sends the garbled tables and the index j to Bob,
   Alice sends the keys for her input wires to Bob by considering the binary representation of her value a_i,
   Bob evaluates the circuit by considering the binary representation of his value b_j.

   If a match is found, Alice adds the value to the result set and will skip further comparisons with the current indices (in Step 1 the lists storing the sets of each party are sorted to allow for a greater chance of finding a match early and thus avoiding useless comparisons, which would certainly be more expensive than sorting a list);
5. finally, Alice sends an "OK" message to Bob to let him know that the computation is over.

My floating point to bit list conversion function is heavily based on a function which is included in the Python standard library.

I made the following modifications to the code implementing Yao's protocol to augment it with the free XOR optimization [KS08]:

- for each each circuit, Alice generates a random value R with length equal to the length of the keys used in the given implementation of Yao's protocol (yao.GarbledCircuit.__init__, line 214);
- for each wire w that is not the output wire of a XOR gate, Alice generates randomly the key k_0^w corresponing to it having a value of 0, and then generates the other one according to the formula:
  $$k_1^w = k_0^w \oplus R$$
  (yao.GarbledCircuit._gen_keys);

- for each XOR gate with input wires x, y and output wire z, Alice generates both keys for the output wire deterministically:

  $$k_0^z = k_0^x \oplus k_0^y$$
  $$k_1^z = k_0^z \oplus R$$

  (yao.GarbledCircuit._gen_keys);

- similarly, Alice generates the p-bits (a.k.a. signal bits) randomly for all wires except for the ones which are output wires of a XOR gate, for which the p-bit should be set to be equal to the XOR of the p-bits of the input wires (yao.GarbledCircuit._gen_pbits);

- when evaluating a XOR gate with input wires x, y with keys k^x, k^y, encrypted result bits b^x, b^y and output wire z, Bob avoids the usual symmetric key decryptions and computes instead:

  $$k^z = k^x \oplus k^y$$
  $$b^z = b^x \oplus b^y$$

  (yao.evaluate).

It is easy to see that this way of computing the keys and result bits is correct given how Alice generated the keys and the p-bits. A formal proof of correctness and security of this optimization of Yao's protocol is given in the original paper [KS08].

4 How to run it

The code was tested using Python 3.8.10 on Ubuntu 20.04. I would suggest running it on Python 3.8+. The dependencies are the same as the ones required by the package implementing Yao's protocol, plus some modules from the Python standard library (multiprocessing, struct, base64). An optional dependency is the tqdm package.

The main entry point of the program is psi.py, which you can run using your python intepreter with e.g. `python3.8 psi.py` .

Running the script with the `-h` switch displays an help message which illustrates how to use the script and how the different arguments work.
Restating what is written in the help message:

- the first argument should be the name of the party you want to run (`alice` or `bob`); you should open another terminal window and run the script as the other party to proceed with the computation.

  A special value for this argument is `test`, which runs both parties in the same terminal window and checks the result obtained with the set intersection algorithm based on Yao's protocol against the set intersection computed normally;

- the second argument should be the party's set enclosed in double quotes and braces, with numbers that can be interpreted as 32-bit floats specified in the standard Python notation, e.g. `"{10, 8.88, 200.33e6}"`;

- if you are using the test mode, you should specify two sets, the first for Alice and the second for Bob, separated by a space;

- with the `-o` switch you can choose the output mode between:
  - "minimal": just prints the final result;
  - "info": prints some additional information and a progess bar if you have tqdm;
  - "full": also writes information about each round of OT in the files "ot_Alice.txt" and "ot_Bob.txt" and the garbled tables of each garbled circuit produced by Alice in the file "tables.txt". These files are placed in the directory named "output".

4.1 How to read the circuit tables

If you use the script with the "full" output mode, the tables for each of the circuits (the circuit is always the same but the keys and p-bits vary each time) will be available in the file "tables.txt" in the directory "output".
For each circuit you can read:

- in the first line, the p-bits (a.k.a. signal bits) associated to each of the wires;
- for each gate in the circuit, the garbled table of the given gate:
  - XOR gates do not have a table as it is not needed nor generated because of the free-XOR optimization;
  - for the other binary gates, each line of each table is in the following format:

    $$[e^a, e^b] : [w^a, v^a][w^b, v^c]([w^c, v^c, e^c])$$

    where w^a, w^b, w^c are integers corresponding to the IDs of wires a, b and c (wires a and b are the input wires of the gate whereas wire c is the output wire),
    v^a, v^b and v^c are the real input and output values on the respective wires,
    e^a, e^b and e^c are the external values on the respective wires, that is $e^x = v^x \oplus p^x$ where p^x denotes the p-bit associated to wire x

    for example, a NOR gate with input wires 1 and 2 and output wire 3 which pbits are described by the following dictionary:

    ```
    P-BITS: {1: 0, 2: 1, 3: 1}
    ```

    produces the following table:

    ```
    GATE: 3, TYPE: NOR
    [0, 0]: [1, 0][2, 1]([3, 0], 1)
    [0, 1]: [1, 0][2, 0]([3, 1], 0)
    [1, 0]: [1, 1][2, 1]([3, 0], 1)
    [1, 1]: [1, 1][2, 0]([3, 0], 1)
    ```

    where for each row:

    - ◇ in the first two columns are the external values of the input wires;
    - ◇ in the third column is the ID of the first input wire;
    - ◇ in the fourth column is the real value of the first input wire, that XORed with its p-bit produces the external value in the first column;
    - ◇ in the fifth column is the ID of the second input wire;
    - ◇ in the sixth column is the real value of the second input wire, that XORed with its p-bit produces the external value in the second column;
    - ◇ in the seventh column is the ID of the output wire;
    - ◇ in the eighth column is the real value of the output wire, that is the NOR of the real input values, i.e. the value in the fourth column NOR the value in the sixth column;
    - ◇ in the ninth column is the external value of the output wire, that is the value in the eighth column XORed with the p-bit of the output wire.

References

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz, editors, *Automata, Languages and Programming*, pages 486–498, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[RR20] Olivier Roques and Emmanuelle Risson. garbled-circuit. `https://github.com/ojroques/garbled-circuit`, 2020.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th Annual Symposium on Foundations of Computer Science, Toronto, Canada, 27-29 October 1986*, pages 162–167. IEEE Computer Society, 1986.