Circuit Computation through Yao's Garbled Circuits and Oblivious Transfer Protocol

Design Lab - CS59001

Under Guidance of
Prof. Satrajit Ghosh

Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur

Submitted by

Girish Kumar

19CS30019

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. The inputs from Alice and Bob will consist of sets of integers.
The supported function are:

1. Add
2. Bool
3. Compare
4. Max
5. Min
6. Millionaire
7. Nand
8. Smart

In Each of the above options, we can select which circuit we want to compute. The circuit used by Yao's protocol in order to compute the function will be dynamically build at each Alice connection to Bob. It will depend on the function we want to compute.

In our model, two parties 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.

We are prompting the user Alice to select a function from above to compute, then prompting him to select a subcircuit/subfunction to compute then asking the Alice to provide his inputs. Alice will send its inputs and the circuit to the bob who acts as the evaluator of the circuit by providing his input before evaluating the circuit.

2 Installation

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

- **ZeroMQ** for communications
- **Fernet** for encryption of garbled tables (mode CBC) - from cryptography
- **SymPy** for prime number manipulation
- **pickle** already with python3.*

To install all dependencies: ou8

$ pip 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 alice.

Alice will print her inputs alongside the corresponding outputs from the evaluator Bob. Alice does not know Bob's inputs.

4 Architecture

- **main.py** implements Alice side, Bob side
- **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.

1. **Main.py :** contains the following classes
   a. **YaoGarbler**: This is an abstract base class that serves as the foundation for the Yao protocol implementation. It initializes the circuit and creates a garbled version of it, which is used for secure evaluation.
   b. **Alice**: This class inherits from **YaoGarbler** and represents the party (Alice) who creates the garbled circuit. Alice generates the garbled circuit, sends it to Bob, and then evaluates the circuit for all possible input combinations from both parties to print the final result.
   c. **Bob**: This class represents the party (Bob) who receives the garbled circuit from Alice and evaluates it based on his inputs. Bob sends the evaluation result back to Alice.
   d. Here's a breakdown of the main functionality:
   e. **Circuit Selection**: The main function prompts Alice to choose a JSON circuit file from the circuits directory and then select a specific circuit from that file based on its ID.
   f. **Input Collection**: After selecting the circuit, Alice is prompted to enter her inputs for the circuit wires assigned to her. Similarly, Bob is prompted to enter his inputs for the wires assigned to him.
   g. **Yao Protocol Execution**:
      i. Alice creates the garbled circuit and sends it to Bob along with her encrypted inputs.
      ii. Bob receives the garbled circuit and evaluates it using Alice's encrypted inputs and his own inputs.
      iii. Bob sends the evaluation result back to Alice.

h. **Result Printing**: Alice prints the final result of the circuit evaluation, showing her inputs, Bob's inputs (assumed to follow a specific order), and the output values.

2. **Yao.py**
   a. Circuit Representation: Circuits are represented using a dictionary structure containing gates (**"gates"**), specifying input wires (**"in"**) and output wire (**"id"**) for each gate, along with the gate's logical function (**"type"**).

   b. Garbling (Encryption/Obfuscation):

      i. P-bits (**self.pbits**): Randomly generated bits associated with each wire of the circuit to mask the true values during evaluation.
      ii. Keys **(self.keys)**: Pairs of cryptographic keys are generated for each wire (0-key and 1-key).
      iii. Garbled Tables (**self.garbled_tables**): Each gate is transformed into a garbled table. For each possible combination of input wire values, the output wire's corresponding key is encrypted in a specific way that allows later decryption only if the correct key combination is provided.

   c. **Evaluation**: To evaluate the circuit, one party has the garbled circuit while the other party has its inputs. Through Oblivious Transfer (not fully shown in this code snippet) and careful cryptographic operations, the circuit is evaluated gate-by-gate without revealing either party's private information.

   d. **GarbledGate**

      i. Represents a single gate in the garbled circuit.
      ii. **_gen_garbled_table_not()** and **_gen_garbled_table():** Functions that construct the garbled table for the gate using encryption and the provided logical operator.
      iii. **print_garbled_table():** Decipherable representation of the gate's logic (useful for debugging).

   e. **GarbledCircuit**

      i. Represents the entire garbled circuit.
      ii. **_gen_pbits()**: Generates random p-bits.
      iii. **_gen_keys()**: Generates cryptographic key pairs for wires.
      iv. **_gen_garbled_tables()** : Creates the garbled table for each gate.
      v. **print_garbled_tables()**: Prints clear representations of the circuit (helps with understanding).

f. **encrypt()** and **decrypt()**

    i. Simple symmetric encryption/decryption helpers, likely using Fernet.

g. **evaluate()**

    i. The core circuit evaluation function. It traverses gates, performs decryptions as needed using inputs, and finally determines the output values.

3. **Ot.py**

   Oblivious Transfer (OT) is a fundamental cryptographic primitive that allows one party (Bob) to retrieve a specific piece of information from another party (Alice) **without** Alice learning which piece was chosen, and **without** Bob learning anything about the other pieces of information. This is essential in garbled circuits to protect the privacy of inputs during the circuit evaluation process.

ObliviousTransfer Class

- **__init__(self, socket, enabled=True)**
    - Initializes the OT object with a socket for communication and a flag (`enabled`) to potentially turn OT on or off (for debugging).
- **get_result(self, a_inputs, b_keys) (Alice's Role)**
    - Sends Alice's (garbler's) encrypted inputs to Bob.
    - Iterates over Bob's input wires:
        - Receives the gate ID where OT needs to be performed.
        - If OT is **enabled**, engages in the **ot_garbler** protocol to send encrypted messages.
        - Otherwise (likely for debugging), sends Bob's keys in the clear.
    - Receives the final circuit evaluation result from Bob.
- **send_result(self, circuit, g_tables, pbits_out, b_inputs) (Bob's Role)**
    - Receives Alice's encrypted inputs.
    - Iterates over Bob's input wires:

        - Sends the gate ID to Alice.
        - If OT is **enabled**, engages in the **ot_evaluator** protocol to retrieve the correct key.
        - Otherwise, receives Alice's keys directly.
    - Evaluates the garbled circuit using **yao.evaluate.**
    - Sends the evaluation result to Alice.
- **ot_garbler(self, messages) (Alice's OT Logic)**

    - Implements the sender's (Alice's) side of the Oblivious Transfer protocol. This is based on Nigel Smart's "Cryptography Made Simple" approach with Diffie-Hellman key exchange.
    - Generates cryptographic values and engages in an exchange with Bob to ensure Bob receives one of the two messages obliviously.
- **ot_evaluator(self, b) (Bob's OT Logic)**

    - Implements the receiver's (Bob's) side of the Oblivious Transfer protocol.
    - Bob uses his input bit (`b`) to cryptographically select one of the messages sent by Alice without revealing his choice.
- **ot_hash(pub_key, msg_length)**

    - A hash function used within the OT protocol to derive keys.

4. **Util.py**

   It focuses on establishing ZMQ sockets for communication, implementing a prime group for cryptographic operations, and providing helper functions. Here's a breakdown of the key components:

   1. Socket Classes

   - **Socket**: The base class for handling ZMQ socket communication. It provides core methods:

     - **__init__(self, socket_type):** Initializes the socket with the specified ZMQ type.
     - **send(self, msg):** Sends a Python object as a message.
     - **receive(self):** Receives a Python object as a message.
     - **send_wait(self, msg):** Sends a message and waits for a reply.

    - **poll_socket(self, timetick=100)**: A generator-based function to efficiently check for incoming messages without indefinite blocking. It handles **KeyboardInterrupt f**or graceful termination.
- **EvaluatorSocket**: Inherits from `Socket` and is configured as a ZMQ REP (Reply) socket. This type expects requests and sends replies. The endpoint it binds to allows connections from garblers.

- **GarblerSocket**: Inherits from **Socket** and is configured as a ZMQ REQ (Request) socket. This sends requests and expects replies. It establishes a connection to the evaluator's endpoint.

2. Prime Group

- **next_prime(num)**: Finds the next prime number greater than `num`. Uses the **sympy** library for prime number calculations.

- **gen_prime(num_bits)** Generates a random prime number of the specified bit length.

- **xor_bytes(seq1, seq2):** Performs a byte-wise XOR operation between two byte sequences.

- **bits(num, width)**: Converts a number into a list of its binary representation (with padding).

- **PrimeGroup**

    - A class representing a cyclic group of prime order used for cryptographic operations.
    - **__init__(self, prime=None):** The constructor allows providing a prime or generates one randomly.
    - **mul(self, num1, num2)**: Multiplication within the group (modulo the prime).
    - **pow(self, base, exponent):** Exponentiation within the group.
    - **gen_pow(self, exponent):** Calculates the generator raised to a given exponent.
    - **inv(self, num)**: Calculates the modular multiplicative inverse of a number within the group.
    - **rand_int(self):** Generates a random integer within the group's range.

- **find_generator(self)**: Finds a generator for the prime group. This is important for cryptographic protocols like Diffie-Hellman.

3. Helper Functions

- **parse_json(json_path)**: A simple utility to load a JSON file.

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.

6 Output Example

First we run the bob as evaluator in 1st terminal using the command below

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

Now in 2nd terminal we will run alice side which acts as the circuit creator using the command below

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

Both alice and bob will provide their input based on the chosen circuit:

Here there is an example of the alice side on and boolean circuit of OR gate.

```
python main.py alice
Available circuit files:
1. add.json
2. bool.json
3. cmp.json
4. max.json
5. million.json
6. min.json
7. nand.json
8. smart.json
Enter the number of the circuit file: 2
Available circuits in bool.json:
1. AND gate
2. OR gate
3. NOT gate
4. XOR gate
5. OR gate
6. NOR gate
7. NAND gate
8. XNOR gate
9. A implies B
10. AA' complement
11. A+A' complement
Enter the number of the circuit: 2
Enter Alice's inputs for circuit OR gate:
Input for wire 1: 1
======== OR gate ========
  Alice[1] = 1  Outputs[3] = 1
```

First alice will choose a function to compute, then alice will choose a circuit inside the function chosen. Then alice will be prompted to provide his inputs based on the number of inputs in the circuit.

```
python main.py bob
Enter Bob's inputs:
Input for wire 2: 0
Received OR gate
```

After this in bob's terminal, bob is prompted to provide its input. Then finally bob will evaluate th value of the circuit.
Alice will print his inputs and the output of the circuit.

Similarly other functions and circuits can be computed.

References:

1. two-party secure function evaluation (https://en.wikipedia.org/wiki/Secure_two-party_computation) using4

2. Yao's garbled circuit https://en.wikipedia.org/wiki/Garbled_circuit)
3. Github Repository : https://github.com/ojroques/garbled-circuit.