Secure Two-Party Computation Assignment

October 9, 2017

1 Introduction

Yao's garbled circuits protocol allows two parties to perform an arbitrary computation without revealing their inputs to each other. In this assignment, your task will be to implement the garbled circuit protocol.

For our purposes, the computation is expressed as a feed-forward boolean circuit. To keep things simple, we only consider gates with 2 inputs and 1 output wire. We also simplify things by assuming that only one party provides input. Gates can be connected to each other (the output wire of one gate can be used as the input wire to other gates) to create a whole circuit.

In the next section we will explain the garbled circuit protocol. It naturally consists of two components, one for each party (the circuit generator, Alice; and the circuit evaluator, Bob). In this assignment you will need to implement the code for each of these roles.

After the protocol description, we'll describe the particular JSON-based file format you will need to use to take a boolean circuit as input, and a variation of that format you will need to use to produce (or consume) garbled circuits, depending on the part of the protocol you are implementing.

After the file format, we'll describe the Python skeleton code you are provided with, which contains tools for parsing and evaluating circuits, along with several example files and utilities you can use for testing.

2 Garbled Circuits Protocol

The garbled circuits protocol works as follows:

1. Alice (the circuit generator), first generates secret keys (for a special symmetric encryption scheme described below) for each wire of the circuit. Each key (called a wire label) is associated with one of the possible values for the wire (0 or 1, since we are considering boolean circuits). For example, for a wire w1, Alice will create a key for each of these values (w1.key0, w1.key1).

2. After that, Alice generates a garbling table for each gate based on the gate's input and output wire labels. For example, suppose gate g1 is an AND gate with two input wires (w1 and w2) and one output wire (w3). Then Alice generates the table for g1 in following way:

In label 1	In label 2	Garbling Table
w1.key0	w2.key0	$\mathsf{Enc}_{\text{w1.key0}}(\mathsf{Enc}_{\text{w2.key0}}(\text{w3.key0}))$
w1.key0	w2.key1	$\mathsf{Enc}_{\text{w1.key0}}(\mathsf{Enc}_{\text{w2.key1}}(\text{w3.key0}))$
w1.key1	w2.key0	$\mathsf{Enc}_{\text{w1.key1}}(\mathsf{Enc}_{\text{w2.key0}}(\text{w3.key0}))$
w1.key1	w2.key1	$\mathsf{Enc}_{\text{w1.key1}}(\mathsf{Enc}_{\text{w2.key1}}(\text{w3.key1}))$

3. After generating this table, Alice randomly permutes the rows of the table, and sends it to Bob. Alice also sends Bob the wire labels corresponding to values of her inputs.

4. Finally, Bob will evaluate the whole circuit by decrypting each gate one by one. At each gate, Bob will only know one of the wire labels, key0 or key1. Therefore, he will only be able to successfully decrypt one row of the table. For example, if Alice reveals input wire labels w1.key1 and w2.key1 respectively, then only row four of the table will be successfully decrypted; therefore, Bob will only learn w3.key1.

5. Once Bob has evaluated the complete circuit, he will send the decrypted labels for each output wire to Alice. Alice will then match the output with the corresponding wire label.

A special encryption scheme. The garbled circuit protocol makes use of a special encryption scheme, $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$, which is a multi-message secure encryption scheme with a couple of special properties. First, since we need to apply this encryption scheme twice, we must require that the ciphertext size is not too much larger than the input message size. More specifically, if the key size is n bits, then messages up to size $\ell \leq 3n$ can be encrypted, and the resulting ciphertexts are of size $\ell + 2n$, i.e. $\mathsf{Enc}_k : \{0,1\}^\ell \to \{0,1\}^{\ell+2n}$ where $\ell \leq 3n$.

Second, there must exist a negligible function ϵ such that for every n and message $m \in \{0,1\}^\ell$, we have that

$$\Pr[k \leftarrow Gen(1^n), k' \leftarrow Gen(1^n), c \leftarrow \mathsf{Enc}_k(m) : \mathsf{Dec}_{k'}(c) = \bot] > 1 - \epsilon(n).$$

In other words, the encryption scheme is such that when a ciphertext is decrypted with the incorrect key, then the output is almost always $\bot$.

Such an encryption scheme can be easily obtained using a length-quadrupling pseudorandom function family, $f_s : \{0,1\}^{|s|} \rightarrow \{0,1\}^{4|s|}$. The encryption scheme is given below:

$\mathsf{Gen}(1^n) : k \leftarrow \mathcal{U}_n$.

$\mathsf{Enc}_k(m) : r \leftarrow \mathcal{U}_n$. Output $r \| (0^n \| m \bigoplus f_k(r))$.

$\mathsf{Dec}_k(c)$: Parse c as $c_1 \| c_2$, where $|c_1| = n$. Compute $m_1 \| m_2 \leftarrow c_2 \bigoplus f_k(c_1)$ where $|m_1| = n$. If $m_1 = 0^n$ output m_2. Otherwise, output $\bot$.

Why is this protocol useful? In this simple example, only Alice provides any input. Clearly, Alice could just compute the function over her input herself. Perhaps this is still meaningful if Alice's goal is to "outsource" her computation to Bob if Alice cannot compute very fast! Really this is not two-party computation, but "outsourced" computation. It would seem the only benefit that Alice gets. This could still be useful in practice!

Generalizations. In the more general case, both Alice and Bob can provide inputs. The value of the protocol then is that neither party learns the other's input. For example, computing statistics or training a machine learning classifier on private datasets. As a bonus problem, you could extend the protocol to include *oblivious transfer* so that both parties can provide inputs.

Even after implementing oblivious transfer, the garbled circuit protocol is only secure in the "honest but curious" attacker model. That is, it defends against passive eavesdropping attacks, but not against active tampering. To withstand against even active attacks, the parties can implement *cut-and-choose*.

3 JSON Circuit File Format

We use a JSON-based file format, Simple Circuit JSON , for providing a description of a boolean circuit and input wire values, which your garbled circuit protocol must accept as input. We also use a variation of this JSON-based file format, Garbled Circuit JSON , to describe the interface between Alice and Bob, i.e. to contain the output of the garbled circuit generator. In this section we explain the two file formats in detail.

3.1 Simple Circuit JSON file format

The Simple Circuit JSON file contains a single top-level JSON object, containing a `gates` object and an `inputs` field.

The `gates` object contains one field for each gate in the circuit. The name of each gate can be an arbitrary JSON-representable string. Each gate object contains the following fields:

- `type`, specifies the type of the gate, which can be "AND," "OR," or "XOR."
- `inp`, an array that specifies the two input wires of the gate. The array contains exactly two elements, where each element is the string name of a wire.
- `out`, an array of one element, specifying the output wire of the gate.
- `table`, an array of four elements that specifies the truth table for the gate. Each entry is an integer bit i.e. 0 or 1.

Either the "type" field or the "table" field must be present. If they are both provided, then they must be consistent. Note that the order of the four elements in the truth table is given as 00, 01, 10, 11.

Note that wires are not explicitly represented as objects; a wire is implicitly defined when its name is included in the `inp` or `out` field of some gate. If a wire appears in some `inp` but not in any `out`, then it is an input wire; vice-versa for output wires.

The `inputs` field is an array that specifies the inputs to the circuit. Each entry in it is a key, value pair. The key is the name of the wire and the value is the integer value for each wire, either 0 or 1.

An example Simple Circuit JSON file is shown below:

```
{
  "gates":{
    "g1":{
      "type":"AND",
      "inp":["w1","w2"],
      "out":["w5"],
      "table":[0, 0, 0, 1]
    },
    "g2":{
      "type":"XOR",
      "inp":["w3","w4"],
      "out":["w6"],
      "table":[0, 1, 1, 0]
    }
  },
  "inputs":{
    "w1": 0,
    "w2": 0,
    "w3": 1,
    "w4": 1
  }
}
```

We have included an example Simple Circuit JSON file named `circuit.json`. We have also provided you with *32adder.json*, which computes 32 bit addition of numbers.

3.2 Garbled Circuit JSON file format

The Garbled Circuit JSON file format contains all of the information output by the garbled circuit generator. This includes everything that must be sent from Alice to Bob, i.e. the garbling table for each gate, as well as the wire labels corresponding to Alice's input values. The Garbled Circuit JSON file also contains additional diagnostic information (namely the mapping between wire labels and wire values). This information would would not actually be sent to Bob (since it would reveal Alice's input), but it will be useful for checking that your generator works correctly.

To keep things simple, we reuse as much of the structure as possible from the Simple Circuit JSON format. At the top level, the main JSON object is divided into the `gates` and `inputs` objects, as well as a `labels` object. The `gates` JSON object is further divided into a list of gate objects. Each gate object consists of the following fields:

- `type`, specifies the type of the gate, values can be "AND," "OR," or "XOR."
- `inp`, an array that specifies the two input wires of the gate. The array contains exactly two elements, where each element is the string name of a wire.
- `out`, an array of one element, specifying the output wire of the gate.
- `garble_table`, an array of four elements that specifies the garbling table entries of the gate. Each entry is a hexadecimal string of length 160, which represents the 80-byte ciphertext computed according to the Step 2 of the garbled circuit protocol.
- `wire_labels`, a mapping from each wire array of four elements that specifies the garbling table entries of the gate. Each entry is a hexadecimal string of length 160, which represents the 80-byte ciphertext computed according to the Step 2 of the garbled circuit protocol.

The `inputs` field is an object that specifies the input wire labels corresponding to Alice's input to the circuit. Each entry in it is a key, value pair. The key is the name of the wire and the value is the hexadecimal string representing the 128-bit wire label as computed in step 1 of the protocol.

The `labels` field is an object that specifies the both wire labels for each wire. Each entry in it is a key, value pair. The key is the name of the wire and the value is an array of two hexadecimal strings, the first of which contains the 128-bit 0 label, the second of which contains the 128-bit 1 label.

An example of the Garbled Circuit JSON file is shown below:

```
{
  "gates":{
    "g1":{
      "type":"AND",
      "inp":["w1","w2"],
      "out":["w5"],
      "garble_table":[
        "9d1a...3354",
        "4b3d...f8ef",
        "1eaf...9b02",
        "923d...be3e"
      ],
    },
    "g2":{
      "type":"XOR",
      "inp":["w3","w4"],
      "out":["w6"],
      "garble_table":[
        "9ea8...65f9",
        "262f...9ec5",
        "987e...177b",
        "0571...4ff8"
      ],
    }
  },
  "inputs":{
    "w1": "3b500391ac11793a7a872cc8817f92e1",
    "w2": "b6d947b216bc74675ca6458ed060c924",
    "w3": "108b8f6260fb4434efe9741b2ef7c89b",
    "w4": "88832adc230360eff3514fe199774fa3"
  },
  "wire_labels":{
    "w1": ["3b500391ac11793a7a872cc8817f92e1","18791201a4d43be6e02e59491746b7b8"],
    "w2": ["b6d947b216bc74675ca6458ed060c924","97684abdefeb6678aa0161d12ea454cc"],
    "w3": ["108b8f6260fb4434efe9741b2ef7c89b","29d3181fb3afec4aafafd3148dca83e3"],
    "w4": ["2b5ba44f376d989ff85a96ef9947a9bf","88832adc230360eff3514fe199774fa3"],
    "w5": ["f7c01eeeabf40831af00b15f051930a3","2b45053afee138724843150ba505f590"],
    "w6": ["d32ccc5aceadffda0f4eb8a94da1c7e0","bd8636e4a35482cca47acb0f28b6f008"]
  }
}
```

The skeleton code a sample garbled circuit file, named *gar_circuit.json*. This represents the same circuit as *circuit.json*, but with the garbling tables and wire labels.

4 Tasks

Your goal is to provide an implementation of the garbled circuits protocol, including the behavior for Alice (the circuit generator) and Bob (the circuit evaluator). We have provided a partial implementation (skeleton code) to start you on your way. The skeleton code also includes a tool for reading and evaluating an input boolean circuit in the Simple Circuit JSON format.

Below we describe the required tasks in detail, in the order we recommend approaching them. You can also search for the string `TODO` in the python files for hints on where to begin.

1. **Generating Cryptographically-secure Random numbers in Python** We have given you three examples of generating random bytes in python. You can find these inside the *util.py* file (see skeleton code explanation). Your task is to explain which one is appropriate for cryptographic schemes (such as shuffling the rows in your garbling table, and generating the wire labels).

   For this task, please also include a comment inside your code explaining the task. You should then use that one in the remainder of your code.

2. **Decryption** We have provided you with an implementation of the length-quadrupling PRF as mentioned in the protocol description, called `util.lengthQuadruplingPRF`. This PRF is based on the AES block cipher, and uses a fixed key size of 128 bits, and returns 512 bits (a 64-byte array). We have also provided you with an implementation of the special encryption routine `Enc`, called `util.specialEncryption`. This function takes in a 128-bit random key and an ℓ-bit message (ℓ is up to 384 bits) and returns an (ℓ+256)-bit ciphertext as a byte array. However, you will need to implement the special decryption routine yourself, based on the protocol description. It should return *null* if decryption fails.

3. **Bob, *The Evaluator* (see `evaluator.py`)** The role of the Evaluator is to read a Garbled Circuit JSON file and evaluate it based on input keys (encryption keys). You can test your evaluator with the included

gar_circuit.json file. We recommended implementing the evaluator before the generator. In the following we explain the steps to create the evaluator:

- Read and parse the garbled circuit file. The skeleton code already creates a python dictionary object by loading the file. For example, `from_json["gates"]` will return the `gates` subobject.
- Evaluate the circuit, gate by gate, in topological order (i.e., starting with the gates that that contain only input wires). For each gate, attempt to decrypt (and decrypt again) each row of the table, using the input wire labels as encryption keys (recall that the special decryption routine will return `None` if the decryption fails because the wrong key is used). Only one row should successfully be decrypted; the decrypted value is the label of the output wire for that gate.
- Finally, return a dictionary mapping the output wire names to their resulting label.

4. **Alice, *The Generator* (see `generator.py`)** The role of the Generator is to read a Simple Circuit JSON file and translate it into a Garbled Circuit JSON file. The output of the generator can then be used by the Evaluator (Bob). In the following we explain the functionality of the generator:

   - Parse the Simple Circuit JSON file. In the skeleton code, the JSON file is already parsed and returned as a python object.
   - For each gate, generate a 128-bit key (a `byte[]` array of length 16) for each wire label, one key for the 0 value and one for the 1 value for each wire. (Step 1 of the protocol)
   - Once the keys each wire are created, generate the garbled table for each gate as explained in Step 2 of the protocol. Don't forget to shuffle the table.
   - For each input value in the Simple Circuit JSON file, pick the corresponding wire label and store this as the input. For example, if w_1 is one of the input wires to the circuit and its value is 0 then pick the key associated to 0 in step 1.
   - Once all wire labels, garbling tables, and input labels are computed, write these to the file as a Garbled Circuit JSON object. For this make sure to follow the format of *circuit.json*. Also, care should be taken that input and output wire names for the gates should remain consistent with the circuit file. Note that the `util.specialEncryption` returns a ciphertext in byte array form, which you must convert to a hexadecimal string before writing it to a Garbled Circuit JSON file.

   We provide a (compiled and obfuscated) version of our reference evaluator, *obfuscated_evaluator.pyz* that you can execute to test the Garbled Circuit JSON files created by your generator. (Of course, if you implement your own evaluator first, as we recommend, then you can use that too.) Instructions for running the reference evaluator can be found in the `README.md` file in the repository.

5. **Bonus Task: Create a multiplication circuit for multiplying n-bit numbers, say $n = 16$**

   (a) Create a circuit file, `multiplication_circuit.json`, implementing n-bit number multiplication.

   (b) Include a description in your report of how you designed the circuit.

   (c) What is the asymptotic size of your circuit, e.g. $O(n^3)$, $O(n^2)$?

   (d) There is an efficient method to implement a multiplication circuit, due to Karatsuba. Please implement a Karatsuba multiplication circuit for $n = 16$ bit numbers, and turn in the circuit file. Please read the following resource that contains an explanation of the Karatsuba algorithm https://en.wikipedia.org/wiki/Karatsuba_algorithm

We will grade this bonus problem by a) checking correctness of your multiplication circuit; and b) using the size of your circuit to assign relative scores. The smallest circuit that is correct will get the greatest number of bonus points!

5 Skeleton Code Explanation

Each code file inside the skeleton code also contains descriptions of their functionality in comments at the top. The `README.md` file contains useful commands you can run.

5.1 How to run the code

Once the code is compiled, you can run:

```
python circuit.py example_circuits/circuit.json
```

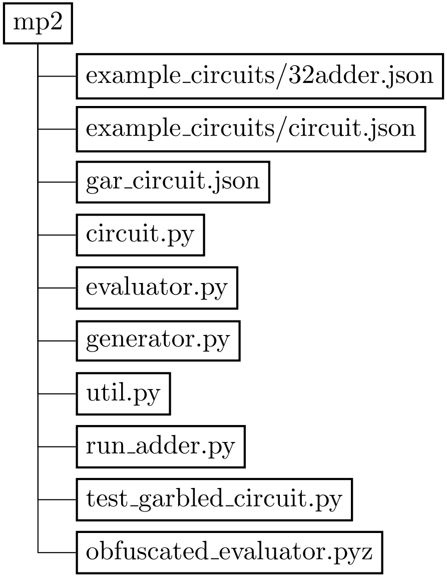
This command will run the *circuit.py* main function with *circuit.json* as input. *circuit.py* contains a simple circuit evaluator, which evaluates a Simple Circuit JSON file.

The files *evaluator.py* and *generator.py* also have main functions. You can run them without any arguments to have them print their usage string. Of course, these files will not run entirely until you complete them.

We have also provided you with an *run_adder.py* script that will help you to change the inputs to the 32-bit adder circuit. When your generator and evaluator are ready, the file *test_garbled_circuit.py* can be used to test your code on a battery of randomly chosen inputs.

5.2 Directory Structure

After cloning the assignment repository, you should have following files in folder:



5.3 Basic Circuits

BooleanCircuit Class This class represents a circuit loaded from a Simple Circuit JSON file, and evaluating it on inputs. It is extended by the garbled circuit generator and evaluator.

- **The constructor** will take a garbled circuit object as an input (loaded from a JSON file), and initialize the *self.gates*, *self.wires*, *input_wires*, *self.output_wires* objects. It will also topologically sort the wires, such that *self.sorted_gates* contains a list of gate ids in sorted order.
- the **evaluate** method takes as input a dictionary mapping input wire ids to values (either 0 or 1). It returns a dictionary mapping output wire ids to values. As a side effect, *wire_values* will also contain the values of all the internal wires.

5.4 Evaluator

The GarbledCircuitEvaluator class. This class inherits from *BooleanCircuit*, and will contain the logic to parse and evaluate the Garbled Circuit JSON file as explained in Section 4.3. In following, we have provided the explanation for each function of this class that are related to the evaluator only.

- **The constructor** should take a garbled circuit JSON object as an input, and store all the garbled tables. The skeleton code already calls the superclass constructor to load the gates and wires. It remains for you to handle the *inputs* and *garble_table* fields.
- the **garbled_evaluate** method takes as input a dictionary mapping each input wire ids to a wire labels. It should evaluate all the garbled gates based on these inputs. In order to evaluate each gate, you must attempt to decrypt each row entry using the input wire labels. It returns a dictionary mapping output wire ids to output wire labels.

5.5 Generator

The GarbledCircuitGenerator class. This class should contain the logic to parse the Simple Circuit JSON file and generate a Garbled Circuit JSON file as explained in Section 4.4. In following, we have provided an explanation for each function of this class that are related to the generator only.

- **The constructor** will take a simple circuit JSON object as input, and generate a garbled circuit. First all the wire labels should be created and after that these labels should be used to compute gate tables.
- **output** is already written for you, and outputs a JSON file containing the circuit object as well as the *self.wire_labels* and *self.garble_table* objects.

5.6 Utility functions

util module. This module contains a few helper functions, related to generating keys and encryption/decryption.

- **specialEncryption**, takes a 128-bit key and up-to-384-bit message and computes an encryption of it.
- **specialDecryption** (incomplete, for you to finish), takes a 128-bit key and a ciphertext, and attempts to decrypt it, returning `null` if the wrong key is used.
- **generate_key** (incomplete, for you to finish), generates a random 128-bit key. To finish this, you need to complete the *random_bytes* function by choosing an appropriate generator.

6 Extensions and Optimizations

For bonus points, implement one or both of the following optimizations discussed in class, the Free XOR trick http://www.cs.toronto.edu/~vlad/papers/XOR_ICALP08.pdf and the Half Gates trick https://eprint.iacr.org/2014/756. Since we will test these separately (the included *gar_circuit.json* example file and *obfuscated_evaluator.pyz* tool will not be compatible), please include the optimized version as additional files named *optimized_evaluator.py* and *optimized_generator.py*.

You will need to come up with a modified schema for the JSON files to support these optimizations. So, please include documentation (in the form of comments in your code) on how your garbled circuit schema was changed.