

Secure Two-Party Computation Assignment

April 19, 2017

1 Introduction

The garbled circuit 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 that Java 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 circuit 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	$\text{Enc}_{w2.\text{key}0}(\text{Enc}_{w1.\text{key}0}(w3.\text{key}0))$
w1.key0	w2.key1	$\text{Enc}_{w2.\text{key}0}(\text{Enc}_{w1.\text{key}1}(w3.\text{key}0))$
w1.key1	w2.key0	$\text{Enc}_{w2.\text{key}1}(\text{Enc}_{w1.\text{key}0}(w3.\text{key}0))$
w1.key1	w2.key1	$\text{Enc}_{w2.\text{key}1}(\text{Enc}_{w1.\text{key}1}(w3.\text{key}1))$

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, (Gen, Enc, 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. $\text{Enc}_k : \{0, 1\}^\ell \rightarrow \{0, 1\}^{\ell+2n}$ 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

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

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

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:

$\text{Gen}(1^n) : k \leftarrow_n.$

$\text{Enc}_k(m) : r \leftarrow_n.$ Output $r \parallel (0^n \parallel m \oplus f_k(r))$.

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

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.

Since each gate contains an explicit truth table representing its functionality, the “type” field can be ignored and treated just as a comment.

Note that wires are not explicitly represented as objects; a wire is implicitly defined when it 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 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.
- `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.

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"
    ],
    "table": [
        "9d1a...3354",
        "4b3d...f8ef",
        "1eaf...9b02",
        "923d...be3e"
    ]
},
"gate": {
    "type": "XOR",
    "inp": [
        "w3",
        "w4"
    ],
    "out": [
        "w6"
    ],
    "table": [
        "9ea8...65f9",
        "262f...9ec5",
        "987e...177b",
        "0571...4ff8"
    ]
},
"inputs": {
    "w1": "3b500391ac11793a7a872cc8817f92e1",
    "w2": "b6d947b216bc74675ca6458ed060c924",
    "w3": "108b8f6260fb4434efe9741b2ef7c89b",
    "w4": "88832adc230360eff3514fe199774fa3"
},
"wires": {
    "w1": {
        "0": "3b500391ac11793a7a872cc8817f92e1",
        "1": "18791201a4d43be6e02e59491746b7b8"
    },
    "w2": {
        "0": "b6d947b216bc74675ca6458ed060c924",
        "1": "97684abdefeb6678aa0161d12ea454cc"
    },
    "w3": {
        "0": "108b8f6260fb4434efe9741b2ef7c89b",
        "1": "29d3181fb3afec4aafafd3148dca83e3"
    },
    "w4": {
        "0": "2b5ba44f376d989ff85a96ef9947a9bf",
        "1": "88832adc230360eff3514fe199774fa3"
    },
    "w5": {
        "0": "f7c01eeeeabf40831af00b15f051930a3",
        "1": "2b45053afee138724843150ba505f590"
    },
    "w6": {
        "0": "d32ccc5aceadffda0f4eb8a94da1c7e0",
        "1": "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 java files for hints on where to begin.

1. **Generating Cryptographically-secure Random numbers in Java** We have given you three examples of generating random bytes in Java. You can find these inside the `Util.java` 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 $(\ell+256)$ -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.java`) 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. For documentation on the `JSONObject` API, see <https://stleary.github.io/JSON-java/org/json/JSONObject.html>. The skeleton code already creates a `JSONObject` by loading the file. You will need to use the `JSONObject` API to traverse the fields of this object. For example, `circuit.get("gates")` will return the `gates` subobject.
- Evaluate the circuit, gate by gate, in topological order (i.e., starting with the gates 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 `null` 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, print the labels of the output wires of the circuit, as hexadecimal strings.

Note that the evaluator does not need to do anything with the `labels` field of the garbled circuit file.

The exact format of how you print the output wire labels does not much matter. When we run your program to grade it, we will simply use “grep” to look for the presence of the correct wire label in the output.

4. **Alice, The Generator** 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, we provide examples of using the `org.json.JSONObject` library to parse a JSON file into a java object and how to iterate over the 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.
 - 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 may have to convert to a hexadecimal string before writing it to a **Garbled Circuit JSON** file.

We provide a (compiled bytecode) version of our reference evaluator that you can use 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 skeleton code.

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

- Create a circuit file, `multiplication_circuit.json`, implementing n -bit number multiplication.
- Include a description in your report of how you designed the circuit.
- What is the asymptotic size of your circuit, e.g. $O(n^3)$, $O(n^2)$?
- 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

The best way to build the skeleton code is by using Apache Maven. In order to install it, please follow the install instructions given in this link: <https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html>.

Once you have installed Apache Maven, you can run

```
$ mvn compile
```

This command will compile the complete source code. We have already included all the libraries required to run the code in `pom.xml` file. This is the default file that Maven will use to download the required libraries. In order to add further libraries you can add it as dependency inside `pom.xml`, as given in this link: <http://www.tech-recipes.com/rx/39256/add-dependencies-to-maven-pom-xml-file/>

Once the code is compiled, you can run:

```
$ mvn exec:java -Dexec.mainClass="com.applied_crypto.Circuit" \
-Dexec.args="./circuit.json"
```

This command will run the *Circuit* main function with `circuit.json` as input. *Circuit* is a simple circuit evaluator, which evaluates a [Simple Circuit JSON](#) file.

Note that you can use above command by changing the parameters properly. Normally the command has two parameters:

-Dexec.mainClass= address of the main class, you may change it with your main class file.

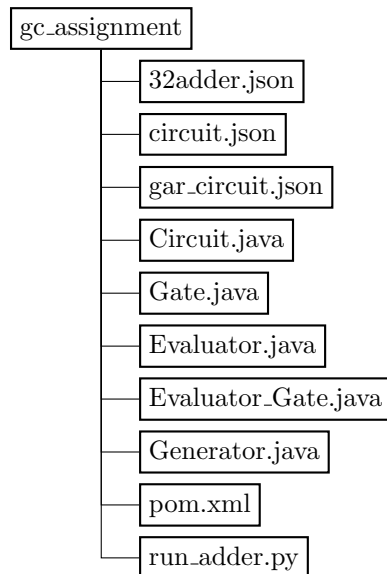
-Dexec.args= any arguments needed to pass to the main file.

Please note that on first run Maven is going to install a lot of required libraries and plug-ins. Just don't panic!

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.

5.2 Directory Structure

After decompressing `gc_assignment.zip` you should have following files in folder:



5.3 Evaluator

Evaluator Class This class 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** will take garbled circuit JSON as an input and converts it into the circuit JSON object.
- **create_circuit**, will parse the circuit JSON object and store all the gates in it along with their inputs and outputs.
- **load_inputs**, will parse the circuit JSON object and store all the inputs specified in it.
- **evaluate_circuit**, will evaluate all the gates based on the inputs loaded in load_inputs. In order to evaluate the particular gate, you have to decrypt its table entries using input wire labels. We have provided the helper function in Util.java, which decrypts all table entries and returns the entry with the tag.

This class also contains a static void main method that parses the input filename from the command line arguments.

5.4 Generator

Generator 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 the explanation for each function of this class that are related to the generator only.

- **The constructor** will take simple circuit JSON as an input and converts it into the circuit JSON object.
- **parse_circuit** will parse the circuit JSON object. First all the wire labels should be created and after that these labels should be used to compute gate tables. Finally, this function will write the gabled circuit into a JSON file.

This class also contains a static void main method that parses the input filename from the command line arguments.

5.5 Helper

Util This class contains a few helper static functions.

- **print_key**, **print_table**, and **print_wire**, print hexadecimal versions of `byte[]` arrays (or `byte[][]` arrays)
- **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.

5.6 Code Evaluation

As mentioned earlier, we have provided the simple circuit evaluator. You can run it using the following command:

```
$ mvn compile
$ mvn exec:java -Dexec.mainClass="com.applied_crypto.Circuit" \
-Dexec.args="./circuit.json"
```

This will parse the input JSON file and print the output of the circuit at the command line. We will evaluate your implementations in the same way.

Evaluator.java should take a **Garbled Circuit JSON** file as input, and prints the output wire labels in hexadecimal to the command line.

Generator.java should take a **Simple Circuit JSON** file as input, and outputs the garbled version of that circuit (the output filename should simply be the input filename, with “gar-” prepended to it).