Lecture Notes for Cryptographic Computing
5. Garbled Circuits

Lecturers: Claudio Orlandi and Peter Scholl, Aarhus University

October 23, 2023

The topic of this week is "garbled circuits" and how they can be used (together with oblivious transfer), to achieve passively secure two-party computation with only constant rounds, aka "Yao's protocol" (as opposed to the BeDOZa-protocol, where the number of rounds was proportional to the circuit depth).

The material presented this week is covered in [HL10, Chapter 3], but we are also going to use some of the abstractions presented in [BHR12].

1 Garbled circuits

A *garbled circuit* is a cryptographic primitive that allows to *evaluate encrypted functions on encrypted inputs* while only revealing the output. Garbled circuits were first introduced by Andrew Yao in the 80s.

Garbled circuits allow to perform secure computation in *constant rounds*. In particular Yao's protocol (combined with a 2-message OT protocol) uses only two rounds. This is crucial in some applications as it allows a user Alice to publish some kind of "encryption" of her input, and then anyone else can send her a single message that allows her to recover the result of the computation.

1.1 Abstract Description of GCs - Garbling Schemes

We follow the definition of garbled circuits by Bellare et al. [BHR12]. A *Garbling Scheme*[1] is defined by a tuple $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ such that:

- The garbled circuit generation function Gb is a randomized algorithm that on input a security parameter 1^k and the description of a Boolean circuit $f : \{0,1\}^n \to \{0,1\}$, (here n and $|f|$ must be polynomially bounded in k) outputs a triple of strings (F, e, d) representing the *garbled circuit*, the *encoding information* and the *decoding information*.
- The algorithm ev allows to evaluate the "plaintext" circuit f i.e., $\mathsf{ev}(f, x) = f(x)$.
- The encoding function En is a deterministic function that uses e to map an input x to a *garbled input* X. A very useful kind of garbling schemes for secure 2PC are the so-called *projective* schemes, where $e = \left(\{K_i^0, K_i^1\}_{i\in[1..n]}\right)$ and the garbled input X is simply $\{K_i^{x_i}\}_{i\in[n]}$ i.e., the encoding information is a set of n pairs of keys, one for each possible value for each bit of the input, and the garbled input X is simply a subset of those strings.
- The *garbled evaluation* function Ev is a deterministic function that evaluates a garbled circuit F on a garbled input X and produces a garbled output Z'.
- The decoding function De, using the decoding information d, decodes the garbled output Z' into a plaintext output z. In *projective schemes* d simply consists of two strings[2] i.e., $d = (Z_0, Z_1)$ and De outputs $z = 0$ if $Z' = Z_0$, $z = 1$ if $Z' = Z_1$ or $\perp$ if $Z' \notin \{Z_0, Z_1\}$.

[1] Throughout this lecture note, we use "Garbling Scheme" for the algorithm that creates and evaluates Garbled Circuits.
[2] In general, two strings for each output bit.

A basic property we will always ask from garbling schemes is correctness.

Definition 1 (Correctness)**.** *Let $\mathcal{G}$ be a garbling scheme. We say that $\mathcal{G}$ enjoys* correctness *if for all $f : \{0,1\}^n \to \{0,1\}$ and all $x \in \{0,1\}^n$ the following probability*

$$\Pr\left(\mathsf{De}(d, \mathsf{Ev}(F, \mathsf{En}(e, x))) \neq f(x) : (F, e, d) \leftarrow \mathsf{Gb}(1^k, f)\right)$$

is negligible in k.

Intuitively, Definition 1 says that evaluating the garbled circuit on the garbled input and then decoding the result should produce the same result as evaluating the plaintext circuit on the plaintext input.

1.2 Passive 2PC based on Projective Garbling Schemes

Projective garbling schemes can be used for two-party computation in the following way:

0. Setup: Alice and Bob have inputs $x, y \in \{0,1\}^n$ and want to compute $f(x, y)$ where $f : \{0,1\}^{2n} \to \{0,1\}$ is a circuit where the first n input bits are controlled by Alice and the second n are controlled by Bob. It will be useful to write $e = (e_x || e_y)$ i.e., to divide the "input enconding information" into a part dedicated to Alice's input and a part dedicated to Bob's input.

1. Garble: Bob runs $(F, (e_x || e_y), d) \leftarrow \mathsf{Gb}(1^k, f)$ and sends F to Alice.

2. Encode Bob's Input: Bob runs $Y \leftarrow \mathsf{En}(e_y, y)$ and sends Y to Alice.

3. Encode Alice's Input: Alice and Bob run a secure two party computation, where Bob inputs e_x, Alice inputs x, and Alice learns $X \leftarrow \mathsf{En}(e_x, x)$. As we use a projective garbling scheme, this means that Bob knows n pair of strings $e_x = \{K_0^i, K_1^i\}_{i\in[1..n]}$, Alice has a vector of bits $x \in \{0,1\}^n$ and Alice should learn $X = \{K_{x_i}^i\}_{i\in[1..n]}$. This can be simply done using n OTs: In the i-th OT Bob acts as the sender and inputs K_0^i, K_1^i and Alice inputs x_i. Thanks to the OT Alice learns (only) $\{K_{x_i}^i\}_{i\in[1..n]}$ and Bob learns nothing.

4. Evaluation: Alice computes $Z \leftarrow \mathsf{Ev}(F, (X||Y))$.

5a. Bob Output: Alice sends Z to Bob who outputs $z \leftarrow \mathsf{De}(d, Z)$.

5b. Alice Output: Bob sends d to Alice who outputs $z \leftarrow \mathsf{De}(d, Z)$.

Remark on the output phase: Depending on the application (and whether Alice or Bob is supposed to learn the output) one can use either Step $5a$ or $5b$.

Remark on the number and ordering of messages: We note that messages $1, 2, 5b$ can be sent together. We note also that if one uses a 2-message OT protocol in phase 3, then the protocol can be compressed to two messages. In the first message Alice sends to Bob the first message of the OT protocol. In the second message Bob sends Alice the second message of the OT protocol together with $1, 2, 5b$.

Security of the protocol: The protocol is secure against passive corruption. Consider the protocol where Alice gets the output (using Step $5b$). Simulating the view of Bob is trivial: Alice does not send any message to Bob except for the OT, so you only need to simulate the OT protocols. The simulator in the case of a corrupted Alice works as follows: given the input and output of Alice x and $z = f(x, y)$ (but not y), the simulator garbles the circuit f' that always outputs z. Now the simulator creates a view using a dummy input x and y and includes F', X', Y', d in the view (and also simulates the OTs). Indistinguishability of the real view and the ideal view of the adversary boils down to the following property of garbling schemes.

Definition 2 (Circuit Privacy)**.** *Let $\mathcal{G}$ be a garbling scheme and denote by $struct(f)$ the structure of the circuit computing f. We say that $\mathcal{G}$ enjoys* privacy *if there exists a simulator S s.t., for all $f : \{0,1\}^n \to \{0,1\}$ and all $x \in \{0,1\}^n$, the output of the simulator*

$$(F', X', d') \leftarrow S(1^k, struct(f), f(x))$$

is indistinguishable from

$$(F, X, d) \text{ where } (F, e, d) \leftarrow \mathsf{Gb}(1^k, f), X \leftarrow \mathsf{En}(e, x)$$

except with probability negligible in k.

The previous definition says that a garbled circuit and a garbled input only leak information about the output of the circuit evaluated on the garbled input, and some partial information $struct(f)$ about the "structure" of the circuit computing f. The structure information includes e.g., the number of gates in the circuit, the number of wires, the number of input and output bits and the wiring of the circuit. It can still hide all other information about the input and the function which is being computed.

One can simply think that everything about the circuit is revealed, except what function the individual gates are computing[3]. If one wants to hide even the structure of the circuit, standard transformation can be applied to the circuit (for instance, one can garble the "universal circuit" that takes as input the description of a circuit of size at most n and evaluates it, and consider the actual function to be computed as part of the input x – in this case only an upper bound on the size of the circuit is revealed). This is very useful in a special kind of secure computation usually referred to as *private function evaluation*, where the setting is: Alice knows x, Bob knows f, Alice should learn $f(x)$ and nothing else about f.

1.3 A concrete garbling scheme

Circuit Notation. We use the following notation for a circuit $f : \{0,1\}^n \to \{0,1\}$: the circuit has T wires; the wires w_i with index $i \in [1..n]$ are called the input wires, the wire w_i with index $i = T$ is called the output wire and all other wires w_i with $i \in [n+1, T-1]$ are called internal wires.

For simplicity we assume the circuit is composed only of NAND gates[4], meaning that for all $i \in [n+1, T]$ $w_i = \neg(w_{L(i)} \wedge w_{R(i)})$, where $L : [n+1, T] \to [1..T]$ and $R : [n+1, T] \to [1..T]$ are two functions specifying the *left input wire* and the *right input wire* of i. We require that $L(i) \leq R(i) < i$, that is, the value on each wire is a function of the values of wires with lower indices. Note that the functions L, R are not defined for the input wires, since those values are given as input and not computed from other values.

Underlying Crypto Primitives. We only need a pseudorandom function

$$G : \{0,1\}^k \times \{0,1\}^k \times [1..T] \to \{0,1\}^{2k}.$$

That is, we use the PRF as $G(A, B, i)$ with A, B being two k-bit keys and i being the input. As a minimum requirement, we need that as long as at least one of the two keys is unknown to the adversary, then the output of the PRF is unpredictable (the exact properties that you need from G is the topic of an exercise below).

[3]Depending on the particular garbling scheme used, more information can be leaked, for instance if one uses the free-XOR technique then the garbled circuit leaks the position and the function computed by the XOR-gates

[4]The garbling technique easily generalizes to arbitrary gates and fan-in larger than 2.

Circuit Generation: To generate a garbled circuit (F, e, d) from a Boolean circuit f with T wires, the algorithm Gb does the following:

1. For each wire $i \in [1..T]$ in the circuit: choose two random strings $(K_0^i, K_1^i) \leftarrow \{0,1\}^k \times \{0,1\}^k$. Let T be the index of the output wire: then define $d = (Z_0, Z_1) = (K_0^T, K_1^T)$. If the circuit has n input wires define $e = \{K_0^i, K_1^i\}_{i \in [1..n]}$.
2. For all $i \in [n+1, T]$ define a garbled table $(C_0^i, C_1^i, C_2^i, C_3^i)$ as follows:
   (a) For all $(a, b) \in \{0,1\} \times \{0,1\}$ compute[5]
   $$C'_{a,b} = G(K_a^{L(i)}, K_b^{R(i)}, i) \oplus (K_{\neg(ab)}^i, 0^k)$$
   (b) Choose a random permutation $\pi : \{0,1,2,3\} \to \{0,1\} \times \{0,1\}$ and add
   $$(C_0^i, C_1^i, C_2^i, C_3^i) = (C'_{\pi(0)}, C'_{\pi(1)}, C'_{\pi(2)}, C'_{\pi(3)})$$
   to F.

Encoding: The encoding function $\mathsf{En}(e, x)$ parses $e = \{K_0^i, K_1^i\}_{i \in [1..n]}$ and outputs $X = \{K_{x_i}^i\}_{i \in [1..n]}$.

Evaluate: The evaluation function $\mathsf{Ev}(F, X)$ parses $X = \{K^i\}_{i \in [1..n]}$ and does, for all $i \in [n+1, T]$:

1. Recover $(C_0^i, C_1^i, C_2^i, C_3^i)$ from F;
2. For $j = 0, 1, 2, 3$ compute:
   $$(K'_j, \tau_j) = G(K^{L(i)}, K^{R(i)}, i) \oplus C_j^i$$
3. If there is a unique j s.t., $\tau_j = 0^k$ define $K^i = K'_j$, else abort and output $\perp$.

Output $Z' = K^T$;

Decoding: The decoding function $\mathsf{De}(d, Z)$ parses $d = (Z_0, Z_1)$ and outputs 0 if $Z' = Z_0$, 1 if $Z' = Z_1$ or $\perp$ if $Z' \notin \{Z_0, Z_1\}$.

Here are some intuitions behind the correctness and privacy of the above construction:

- *(Correctness – why is 0^k there?)* During the garbling step, the circuit constructor "encrypts" the output keys of each gate under all four combinations of the input keys; During the evaluation, the circuit evaluator has only two of the input keys (one for each wire) and needs to find the key corresponding to the output wire for the right value. I.e., the evaluator knows that $K^{L(i)} = K_a^{L(i)}$ for $a \in \{0,1\}$ but does not know the value of a (same for the right key and the value of b), but still needs to be able to compute the output key K_c^i with $c = \neg ab$. This is possible thanks to the "redundancy" 0^k which is appended to the encryption, which allows to decide if decryption was successful or not: clearly, when the evaluator decrypts the right entry in the garbled table he will find a value of the right format $(K, 0^k)$ since by construction
  $$G(K_a^{L(i)}, K_b^{R(i)}, i) \oplus C_{\pi^{-1}(a,b)}^i = (K_{\neg(ab)}^i, 0^k).$$
  However, if $\pi(j) = (a', b') \neq (a, b)$ then it will be the case that:
  $$G(K_a^{L(i)}, K_b^{R(i)}, i) \oplus C_j^i = G(K_a^{L(i)}, K_b^{R(i)}, i) \oplus G(K_{a'}^{L(i)}, K_{b'}^{R(i)}, i) \oplus (K_{\neg(a'b')}^i, 0^k)$$
  Intuitively, since G is a pseudorandom function we expect that the last k bits of
  $$G(K_a^{L(i)}, K_b^{R(i)}, i) \oplus G(K_{a'}^{L(i)}, K_{b'}^{R(i)}, i)$$
  are different than 0^k with very high probability when $(a, b) \neq (a', b')$ and therefore the evaluator will notice that the decryption was not successful.

[5] 0^k is a shortcut for the whole zero string of length k.

- *(Privacy – Why is the PRF there?)* At no point is the circuit evaluator (e.g., Alice) allowed to learn both keys for any wire: this is true (by construction) for the input wires, and we argue that it is true also for the internal wires: assume that for some wire i Alice knows $K_1^{L(i)}, K_1^{R(i)}$.[6] For simplicity, assume even that Alice knows π. We must make sure that Alice cannot compute K_1^i: since Alice does not know $K_0^{L(i)}, K_0^{R(i)}$ she does not know at least k bits for all other evaluations of G and therefore we expect all the other C_j^i with $\pi(j) \neq (1,1)$ to be indistinguishable from random in Alice's view (by assumption on G). More precisely, we need that for all i Alice cannot distinguish between

$$(i, K_1^{L(i)}, K_1^{R(i)}, G(K_0^{L(i)}, K_0^{R(i)}, i), G(K_0^{L(i)}, K_1^{R(i)}, i), G(K_1^{L(i)}, K_0^{R(i)}, i)) \tag{1}$$

and

$$(i, K_1^{L(i)}, K_1^{R(i)}, s_1, s_2, s_3) \tag{2}$$

for three random strings $s_i \in \{0,1\}^{2k}$;

- *(Privacy – Why is π there?)* Assume the circuit constructor did not permute the ciphertexts. What would go wrong? During the evaluation of a given gate Alice learns j such that $\tau_j = 0^k$. If Alice knew the permutation for that wire, then Alice would learn that the values of the input wires to that gate are $\pi(j) = (a, b)$!

Remember that we defined privacy in a different way, namely by asking that there exists a simulator that on input $struct(f)$ and $z = f(x)$ (but not f or x) can simulate F, X, d. We can construct the simulator in the following way:

1. Parse $struct(f)$ (the information about f which is leaked via F) as the number of wires T, the number of inputs n and the "wiring functions" L,R.
2. Choose $T+1$ random keys $K^i \in \{0,1\}^k$. Define $X = \{K^i\}_{i \in [1..n]}$ and define $d = (Z_0, Z_1) = (K^T, K^{T+1})$;
3. For all $i \in [n+1, T]$, choose a random $h \in \{0,1,2,3\}$ and three random strings $C_j^i \leftarrow \{0,1\}^{2k}$ for $j \in \{0,1,2,3\}, j \neq h$. Define
$$C_h^i = G(K^{L(i)}, K^{R(i)}, i) \oplus (K^i, 0^k)$$
for all $i \in [n+1, T-1]$ and define
$$C_j^T = G(K^{L(T)}, K^{R(T)}, T) \oplus (K^{T+z}, 0^k)$$

That is, the simulator creates a garbled circuit where there exists a single key for each wire and each garbled table contains a single encryption and 3 random ciphertexts. The gabled gate of the output wire is slightly different since we need to make sure it outputs the right value (defined by the decoding table d). Indistinguishability follows from the properties of the PRF (we will not cover the proof).

Exercise 1. (How (not) to garble?)

The garbling scheme uses a PRF $G : \{0,1\}^k \times \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^{2k}$ which takes two "half" keys of k bits each and an input of n bits and produce an output of $2k$ bits, with the property that equations (1) and (2) above must be indistinguishable for all i.
Let $g : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^{2k}$ be a PRF which only takes a single k bit key (and an arbitrary input). Which of the following proposals is a secure instantiation of G?

1. $G(A, B, i) = g(A, i) \oplus g(B, i)$;

[6] Note that for most internal wires Alice will not know the values corresponding to her keys, in this case $(1,1)$, but for gates connected only to input wires Alice might actually know these values.

2. $G(A, B, i) = g(A, (i, 0)) \oplus g(B, (i, 1))$;
3. $G(A, B, i) = g(A, g(B, i))$
4. $g(A, i) = (\alpha_1, \alpha_2)$, $g(B, i) = (\beta_1, \beta_2)$ (i.e., α_1 is the first k bits of $g(A, i)$, and so on)): $G(A, B, i) = (\alpha_1 \oplus \beta_2, \alpha_2 \oplus \beta_1)$

Try plugging these proposals in (1) above (previous page) and see if you can distinguish it from random (or argue indistinguishability).
Hint : .

Exercise 2. (Mandatory Assignment)

Implement Yao's protocol for the *blood type compatibility* function. You already have implemented OT in the mandatory assignment for last week. You can implement the PRF using SHA-256, and set the length of the wire labels to 128 bits. The class `MessageDigest` in Java is probably the easiest way of using SHA-256. Since the goal of the exercise is to better understand the protocol (not to build a full functioning system), feel free to implement all parties on the same machine and without using network communication. For example, you could implement Alice and Bob as two distinct classes in Java and then let them interact in the following way:

```
m1 = Alice.Choose(x);
m2 = Bob.Transfer(y,m2);
z = Alice.Retrieve(m2);
```

Exercise 3. (Attack!)

Yao's protocol is secure against passive adversaries but if Bob is actively corrupted then the security of the protocol fails miserably. Think adversarially! How would you break Yao's protocol?

1.4 Towards an Actively Secure Protocol

Yao's protocol is only secure against passive adversaries. Active secure variants of the protocol exist, and some of them are fairly efficient[7]. Here we only discuss some simple properties and countermeasures when parties are corrupted.

Actively corrupted Alice. A corrupted Alice, who evaluates the circuit in garbled form, essentially does not have any way of attacking the protocol (if the OT protocol is secure against active adversaries). This is due to the following, very useful property of garbling schemes:

Definition 3 (Authenticity)**.** *Let $\mathcal{G}$ garbling scheme. We say that $\mathcal{G}$ enjoys* authenticity *if for all PPT A, for all $f : \{0,1\}^n \to \{0,1\}$ and all $x \in \{0,1\}^n$:*

$$\Pr[\mathsf{De}(d, A(F, X)) \notin \{f(x), \perp\}]$$

is negligible in k where $(F, e, d) \leftarrow \mathsf{Gb}(1^k, f), X \leftarrow \mathsf{En}(e, x)$.

[7] This would be a good topic for your project!

Intuitively the property says that an adversary on input a garbled circuit F and a garbled input X can only compute the right garbled output or compute something which will be detected as an invalid output. This implies that even an actively corrupted Alice cannot cheat in Yao's protocol. This property is very useful also in other applications of garbling schemes, such as *verifiable delegation of computation* which is described below.

There is an extra technical issue to be taken care of when running with a corrupted Alice: in the protocol as described above, Alice sees the garbled circuit F *before* she has to choose her input to feed it into the OT. This means that Alice could choose her input adaptively as a function of the garbled circuit F. This introduces so-called *selective opening attacks* – roughly speaking, Alice sees a number of ciphertexts and then can ask for the secret keys corresponding to some of these ciphertexts of her choice. Perhaps surprisingly, it is not possible to prove that the other ciphertexts (i.e., the one for which she does not got a secret key for) are still secure using only the standard *chosen-plaintext attack* security property.

This can be fixed by changing the order of the messages in the protocol: instead of first sending F, Y and then running the OT, one should first run the OT and then send F, Y after the OT protocol is completed. In this way Alice cannot choose her input based on F.

Actively corrupted Bob. An actively corrupted Bob can attack Yao's protocol in several ways. The most significant is by encrypting a different function f^* than the one Alice and Bob had agreed upon. Note that Alice has no way of distinguishing between a garbled version of the real function f and a garbled version of a maliciously chosen function f^* with the same structure. This is ensured by the privacy property (see Definition 2) that we used earlier to argue security of the protocol! In particular, f^* could be a function that outputs a value chosen by Bob, or that leaks information about Alice's input.

A solution to this problem is the so-called *cut-and-choose* paradigm: the main idea is to let Bob garble the function f multiple times, say s, with independently chosen randomness, and send $F_1, \ldots, F_s$ to Alice. Now (similarly to what we do in zero-knowledge proofs) Alice chooses at random some of these garbled circuits and Bob has to provide her with the randomness he used to generate them, so that Alice can verify that they are a garbling of the function f.

If Alice does not detect any cheating attempt, she assumes that the remaining unopened circuits are also honestly generated (with some probability) and then Alice and Bob evaluate the remaining circuits.

Depending on the number of evaluated circuits, we have two main classes of cut-and-choose protocols:

"All-but-one" cut-and-choose: In this class of protocols, Alice chooses one index $j \in [1, s]$ at random. Bob sends Alice the randomness that he used to garble all circuits F_i with $i \neq j$. If Alice does not find any "bad" garbling, the protocol proceeds in the same way as the passively secure one from Step 2.

A malicious Bob still has probability $1/s$ to cheat: he can garble a single malicious circuit and hope that Alice does not check that one. This gives only $1 - 1/s$ security, falling short of our goal to achieve security will all but negligible probability (the technical term for this is *covert security*).

"Majority" cut-and-choose: In this class of protocols, Alice chooses a subset $J \subset [1, s]$ of size $|J| \approx s/2$ circuits to be checked, and then the protocol proceeds with evaluating the unopened $s/2$ circuits. If the evaluated circuits output different values, Alice outputs the value output by the majority of the circuits. A simple combinatorial argument shows that the majority of unopened circuits (for large enough s) are honestly generated, given that Alice did not detect any cheating during the cut-and-choose phase. Note that even if Alice notices that Bob cheated she will not complain! That is, if the evaluation circuits do not output the same value, Alice knows that Bob cheated and garbled different functions. But instead of complaining, she completes the protocol choosing the majority output. This is because if Alice complains, Bob could mount the following *selective-failure attack*: Bob garbles two functions f_1, f_2 s.t., they output the same value if the first bit of x is 0 or different values otherwise. Now depending on Alice's behaviour (abort or not) Bob gets to learn one bit of Alice's input.

There are many different protocols in the literature that use this kind of majority cut-and-choose. The main difference between those protocols is in the way they deal with the following problem: now that Alice and Bob evaluate $s/2$ different circuits, we need to make sure that the inputs they use in all

evaluations are the same. If this is not taken care of, then Alice or Bob (depending on who learns the output of the evaluations) can learn the evaluation of the function on up to $s/2$ different inputs, and this is a problem because multiple evaluations of the function $f(x, y_1), \ldots, f(x, y_{s/2})$ potentially reveal more information about x than a single evaluation $f(x, y)$.

There is another kind of *selective failure attack* that a corrupt Bob can perform on the protocol, by using the "wrong" input during the OT phase. Assume that for some position i, Bob inputs to the OT the pair $(m_0, m_1) = (K_0^i, R)$ for some random string R. Now if the i-th bit of Alice's input is equal to 0, Alice learns K_0^i from the OT and can correctly evaluate the garbled circuit. Instead, if $x_1 = 1$, Alice receives a random key from the OT and therefore the Ev procedure will fail, thus Alice will abort. Now, by observing whether Alice aborts the protocol or not, Bob will learn one bit of the input x_i.

2 Exercises

The following exercises ask you to rediscover some of the most useful optimizations for garbled circuits. The main difference between the garbling technique presented before and the optimized garbling techniques in the exercises is that:

- In the basic garbling scheme, we choose two random keys $\{K_0^i, K_1^i\}$ for all wires i;
- In (most of) the optimized schemes, some of the keys corresponding to the wires are not chosen completely at random, but as a deterministic functions of some of the other keys.

Exercise 4. (Free-XOR Technique)

There is a very useful trick that allows to garble and evaluate XOR-gates for free – in practice this can be very significant: with a few optimizations the AES circuit contains up to 80% XOR gates, see `https://homes.esat.kuleuven.be/~nsmart/MPC/`.
To do so, the circuit constructor chooses a random "global difference" $\Delta \in_R \{0,1\}^n$ and, for each wire in the circuit, sets the key corresponding to 1 to be the XOR of the key corresponding to 0 and the global difference i.e., for each wire i, Bob chooses K_0^i at random and then sets $K_1^i = K_0^i \oplus \Delta$. How do we "garble" XOR gates now? How do we evaluate them?

Exercise 5. (Point-and-permute Technique)

In the original garbling scheme Alice needs to decrypt all four ciphertexts to find out the key for the output wire. A different technique, called *point-and-permute*, allows Alice to evaluate a garbled gate using only one decryption. This means that Alice can save 75% of her time! Also, this allows to remove the 0^k from the encryptions, meaning that all garbled tables are halved in size.
Here is the idea explained for a "unary" gate: For all wires in the circuit, Bob adds an extra bit to the two keys K_0, K_1. That is, he appends a random bit b to K_0 and $1 - b$ to K_1. (What is the lsb of K_b now?). Then he prepares the two ciphertexts $C_0 = G(K_0, i) \oplus m_0, C_1 = G(K_1, i) \oplus m_1$ and he sends them to Alice in the order determined by b i.e., $(C'_0, C'_1) = (C_b, C_{1-b})$. Now Alice is given a key $K \in \{K_0, K_1\}$ without knowing which one it is, and decrypts $m = G(K, i) \oplus C_{lsb(K)}$. Verify that Alice is getting the right output i.e., that for both $\alpha \in \{0, 1\}$ it holds that

$$G(K_\alpha, i) \oplus C_{lsb(K_\alpha)} = m_\alpha$$

Can you now generalize this idea for binary gates?

Exercise 6. (Row Reduction Technique)

There is a technique that allows to reduce the number of ciphertexts that Bob needs to send to Alice for each garbled gate by one. The technique works well with the point-and-permute technique from the previous exercise, so do that first.
In this technique, we do not choose the keys for all wires at random. Instead, the keys (K_0^i, K_1^i) for a wire i will be a function of the keys of the input keys for that gate $(K_0^{L(i)}, K_1^{L(i)}, K_0^{R(i)}, K_1^{R(i)})$ and some additional randomness.
Here is the idea for a unary gate – a NOT gate to be concrete (that is the value on wire i is the logical negation of the value on wire j): The input keys K_0^j, K_1^j are like in point-and-permute i.e. $lsb(K_1^j) \neq lsb(K_0^j) = b$, where b is uniformly random. The output keys K_0^i, K_1^i are chosen as follows: one of the output keys is chosen as function of the input keys i.e., $K_b^i = G(K_{1-b}^j, i)$ and the other key is chosen to be compatible with the free-XOR technique i.e., $K_{1-b}^i = K_b^i \oplus \Delta$
If you compute the ciphertexts as before you get:

$$C_0 = G(K_b^j, i) \oplus K_{1-b}^i$$

$$C_1 = G(K_{1-b}^j, i) \oplus K_b^i$$

If you evaluate C_0 you will notice that it has a fixed value and therefore does not need to be sent.
To evaluate the gate on input K^j, Alice computes

$$K^i = C_{lsb(K^j)} \oplus G(K^j, i)$$

Check that Alice gets the right output key. Can you generalize this to binary gates? (The solution has 3 ciphertexts instead of 4).

3 Verifiable Delegation of Computation Using Garbling Schemes

Garbling schemes can be used for verifiable delegation of computation (with preprocessing). The problem of verifiable delegation of computation is the following: a computationally limited client (i.e., a smartphone) wants to delegate the computation of a circuit f on an input x a computation to a powerful server (i.e., some cloud provider). However, the client is afraid that the server would prefer not to spend time and energy computing the right output and produce a wrong output instead. So the client wants a way of verifying that the output z that he receives from the server is such that $z = f(x)$. Clearly the solution is only useful if the time to verify if $z = f(x)$ is *less* than the time needed to compute $f(x)$ from scratch.

Several solutions to this problem exist using cryptographic tools. Garbling schemes can be used to achieve a partial solution, namely delegation with preprocessing. The protocol proceeds as following:

Preprocessing: The client computes $(F, e, d) \leftarrow \mathsf{Gb}(f, 1^k)$ and sends F to the server.

Online Phase: When the input x becomes available, the client generates $X \leftarrow \mathsf{En}(e, x)$ and sends X to the server; the server computes $Z \leftarrow \mathsf{Ev}(F, X)$ and returns Z to the client who outputs $z \leftarrow \mathsf{De}(d, Z)$.

Thanks to the property stated in Definition 3 (Authenticity) the client is sure that if $z \neq \perp$, then $z = f(x)$. As required, the running time of the online phase is independent of the size of the circuit f (in fact both En, De are extremely lightweight functions).

Unfortunately the complexity of the preprocessing (the complexity of running Gb) is proportional to $|f|$ which is why this is only a partial solution. To motivate the application, you could think of the smartphone generating the circuit while plugged to a power source, and then run the online phase of the protocol while running on battery.

References

[HL10] Carmit Hazay, Yehuda Lindell
Efficient Secure Two-Party Protocols
http://lib.myilibrary.com.ez.statsbiblioteket.dk:2048/Open.aspx?id=300348

[BHR12] Mihir Bellare and Viet Tung Hoang and Phillip Rogaway
Foundations of Garbled Circuits
ACM CCS 2012, available at http://eprint.iacr.org/2012/265