1 Secure Computation - Yao’s Garbled Circuits

We want to answer the question of how Alice and Bob can securely compute any function \( f \) over their private inputs \( x \) and \( y \). In the previous lecture we used the Goldreich-Micali-Wigderson (GMW) Protocol to securely compute functions. This was a highly interactive solution, which naturally extended to any number of parties. We are now going to look at Yao’s Garbled Circuits, another technique for securely computing a function. Yao’s technique requires little interaction between Alice and Bob, but only works with two parties. In order to examine Yao’s Garbled Circuit technique, we must first define what Garbled circuits are.

**Definition 1** A Garbling Scheme consists of two procedures, \( \text{Garble} \) and \( \text{Eval} \):

- \( \text{Garble}(C) \): Takes a circuit \( C \) as input and will output a collection of garbled gates \( \hat{G} \) and garbled input wires \( \hat{I}n \) where
  \[ \hat{G} = \{ \hat{g}_1, \ldots, \hat{g}_c \} \]
  \[ \hat{I}n = \{ \hat{i}n_1, \ldots, \hat{i}n_n \} \]

- \( \text{Eval}(\hat{G}, \hat{I}n_x) \): Takes as input a garbled circuit \( \hat{G} \) and garbled input wires \( \hat{I}n \) corresponding to an input \( x \) and outputs \( z = C(x) \)

Now we will outline how Garbling Schemes work.

- Each wire \( i \) in the circuit \( C \) is associated with two keys \( (k^i_0, k^i_1) \) of a secret-key encryption scheme, one corresponding to the wire value being 0 and other for wire value being 1

- For an input \( x \), the evaluator is given the input wire keys \( (k^1_{x_1}, \ldots, k^n_{x_n}) \) corresponding to \( x \). Also for every gate \( g \in C \), it is also given an encrypted truth table of \( g \), which is something we will show later.

- We want the evaluator to use the input wire keys and the encrypted truth tables to uncover a single key \( k^i_v \) for every internal wire \( i \) corresponding to the value \( v \) of that wire. However, \( k^i_{1-v} \) should remain hidden from the evaluator.

In order to implement this we will have to define a special encryption scheme.

**Definition 2 Special Encryption Scheme** : We need a secret-key encryption scheme \((\text{Gen}, \text{Enc}, \text{Dec})\) with an extra property: there exists a negligible function \( \nu(\cdot) \) s.t. for every \( n \) and every message \( m \in \{0,1\}^n \),

\[ Pr[k \leftarrow \text{Gen}(1^n), k' \leftarrow \text{Gen}(1^n), \text{Dec}_k(\text{Enc}_k(m)) = \bot] < 1 - \nu(n) \]
Essentially this is saying if a ciphertext is decrypted using a different or “wrong” key, then answer is always ⊥.

**Construction**: In order to create this special secret encryption simply modify the secret-key encryption scheme discussion in the secret lecture, except instead of encryption \( m \), we encrypt \( 0^n || m \). Upon decrypting we check if the first \( n \) bits of the message are all 0’s; if they aren’t we output ⊥.

2 Garbled Circuits Construction

We are now going to define Garble and Eval for our Garbled Circuit. Let (Gen, Enc, Dec) be a special encryption scheme (as defined above). Assign an index to each wire in \( C \) s.t. the input wires have indices 1, ..., \( n \).

**Garble(\( C \))**:
- For every non-output wire \( i \) in \( C \), sample \( k^i_0 \leftarrow Gen(1^n) \), \( k^i_1 \leftarrow Gen(1^n) \). For every output wire \( i \) in \( C \), set \( k^i_0 = 0 \), \( k^i_1 = 1 \).
- For every \( i \in [n] \), set \( \tilde{in}_i = (k^i_0, k^i_1) \). Set \( \tilde{In} = (\tilde{in}_1, \ldots, \tilde{in}_n) \)
- For every gate \( g \) in \( C \) with input wire (i)

<table>
<thead>
<tr>
<th>First Input</th>
<th>Second Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( k^i_0 )</td>
<td>( k^j_0 )</td>
<td>( z_1 = Enc_{k^i_0}(Enc_{k^j_0}(k^l_{g(0,0)}) ))</td>
</tr>
<tr>
<td>( k^i_0 )</td>
<td>( k^j_1 )</td>
<td>( z_2 = Enc_{k^i_0}(Enc_{k^j_1}(k^l_{g(0,1)}) ))</td>
</tr>
<tr>
<td>( k^i_1 )</td>
<td>( k^j_0 )</td>
<td>( z_3 = Enc_{k^i_1}(Enc_{k^j_0}(k^l_{g(1,0)}) ))</td>
</tr>
<tr>
<td>( k^i_1 )</td>
<td>( k^j_1 )</td>
<td>( z_4 = Enc_{k^i_1}(Enc_{k^j_1}(k^l_{g(1,1)}) ))</td>
</tr>
</tbody>
</table>

Set \( \tilde{g} = \text{RandomShuffle}(z_1, z_2, z_3, z_4) \). Output \( (\tilde{G}, \tilde{In}_x) = (\tilde{g}_1, \ldots, \tilde{g}_{|C|}, \tilde{In}) \)

**Why is the random shuffle necessary?** If we do not randomly shuffle the outputs an Adversary would know information about the combination of \( k^i, k^j \) used to achieve the output just based on the index of the returned value.

**Eval(\( \tilde{G}, \tilde{In}_x \))**:
- Parse \( \tilde{G} = (\tilde{g}_1, \ldots, \tilde{g}_{|C|}) \). \( \tilde{In}_x = (k^1, \ldots, k^n) \)
- Parse \( \tilde{g}_i = (\tilde{g}_1, \ldots, \tilde{g}_4) \)
- Decrypt each garbled gate \( \tilde{g}_i \) one-by-one in canonical order:
  - Let \( k^i \) and \( k^j \) be the input wire keys for gate \( g \).
  - Repeat the following for every \( p \in [4] \):
    
    \[
    \alpha_p = Dec_{k^i}(Dec_{k^j}(\tilde{g}_p^i))
    \]
    if \( \exists \alpha_p \neq \bot \), set \( k^i = \alpha_p \)
- Let \( out_i \) be the value obtained for each output wire \( i \). Output \( out = (out_1, \ldots, out_n) \)
Secure Computation from Garbled Circuits

Let us discuss a plausible approach for securely computing \( C(x, y) \) using Garbled Circuits.

\( A \) generates a garbled circuit for \( C(\cdot, \cdot) \) along with garbled wire keys for first and second input to \( C \). It then sends the garbled wire keys corresponding to its input \( x \) along with the garbled circuit to \( B \). Note, however, that in order to evaluate the garbled circuit on \((x, y)\), \( B \) also needs the garbled wire keys corresponding to its input \( y \).

A possible solution is for \( A \) to send all the wire keys corresponding to \( B \)’s input by using oblivious transfer. Below, we describe the solution in detail.

**Ingredients:** Garbling Scheme (Garble, Eval), 1-out-of-2 OT scheme \( OT = (S, R) \) as defined in previous lecture on secure computation.

**Common Input:** Circuit \( C \) for \( f(\cdot, \cdot) \)

**A’s input:** \( x = x_1, \ldots, x_n \)

**B’s input:** \( y = y_1, \ldots, y_n \)

**Protocol** \( Pi = (A, B) \):

<table>
<thead>
<tr>
<th>( A \rightarrow B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A ) computes ((\hat{G}, \hat{In})) Parse ( \hat{In} = (\hat{in}_1, \ldots, \hat{in}_2n) ) where ( \hat{in}<em>i = (k^i_0, k^i_1) ). Set ( \hat{In}<em>x = (k</em>{x1}, \ldots, k</em>{xn}) ). Send ((\hat{G}, \hat{In}_x)) to ( B )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( A \leftrightarrow B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>For every ( i \in [n] ), ( A ) and ( B ) run ( OT = (S, R) ) where ( A ) plays sender ( S ) with input ((k^{n+i}_0, k^{n+i}<em>1)) and ( B ) plays receiver ( R ) with input ( y_i ). Let ( \hat{In}<em>y = (k^{y1}</em>{y1}, \ldots, k^{yn}</em>{y2n}) ) be the outputs of the ( n ) OT executions received by ( B ).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( B )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B ) outputs ( \text{Eval}(G, \hat{In}_x, \hat{In}_y) )</td>
</tr>
</tbody>
</table>

In order to argue the security of the construction, we use two properties.

**Property 1:** For every wire \( i \), \( B \) only learns one of the two wire keys:

- **Input Wires:** For input wires corresponding to \( A \)’s input, it follows from protocol description.
  - For input wires corresponding to \( B \)’s input it follows from security of OT

- **Internal Wires:** Follows from the security of the encryption scheme

**Property 2:** \( B \) does not know whether the key corresponds to wire value being 0 or 1 (except the keys corresponding to its own input wires).

From this we can notice that \( B \) only learns the output and nothing else. \( A \) does not learn anything (in particular, \( B \)’s input remains hidden from \( A \) due to the security of OT). The full proof of security can be found in [1].
References