Secure Multiparty Computation: Yao's Garbled Circuits and Oblivious Transfer

Elizabeth Yang

October 8, 2018

1 Motivation

Consider the following (frankly, first world) problem. Imagine you and your incredibly rich friend want to determine who is richer. However, you and your friend don't want to leak how much money you actually have to each other. How do you perform this comparison?

The above scenario is exactly what is being described in Yao's Millionaire Problem. While this isn't a very relevant problem for most people, it does speak to a greater security need. How do we perform an arbitrary computation $f(x_1, \ldots, x_n)$ using inputs x_i from multiple parties without (unnecessarily) leaking one party's input to another?

Secure (multiparty) computation seeks to solve this problem. For this lecture, we focus on one specific kind of secure computation called *Yao's garbled circuits* [4]. Part of Yao's setup relies on an *oblivious transfer* protocol [1], which we will describe at the end of the notes.

2 Main Framework

A garbling scheme consists of three algorithms: (1) Garble, (2) Encode, and (3) Eval. Some texts also include a "Decode" algorithm, however we wrap it into Eval. We'll explain these at a high level, and then illustrate them using a concrete example.

- $\text{Garble}(1^n, F) = (GC_F, k)$: Here, n is the security parameter, which represents a benchmark for defining the runtime limits of an algorithm or an adversary. F is a function that is expressed as a circuit of Boolean gates. GC_F is the garbled circuit, while k is a secret key to be used in Encode.
- $\text{Encode}(k, x) = E_x$: Here, k is a key corresponding to a pair of inputs to F, and x is their corresponding output. The ciphertext, E_x, is used in Eval.
- $\text{Eval}(GC_F, E_x) = F(x)$: This step performs the desired computation.

2.1 A Small Example

To explain the construction of a garbled circuit, we start with a small example: an AND gate. Let's say Ginny (the "generator") and Evan (the "evaluator") each have a bit, and denote them by g and e respectively. They want to compute $(g \wedge e)$ without telling each other g and e.

There is a small technicality: if $g = 1$ for instance, then knowing $(g \wedge e)$ will let Ginny uncover e. The same story holds when $e = 1$; Evan would always be able to recover g. Thus, our security guarantee should be that if $g = 0$, Ginny should not learn e, and if $e = 0$, then Evan should not learn g.

Ginny will run Garble and Encode, while Evan will be responsible for Eval. Let G and E denote their actual bits. (We'll use g and e as symbolic variables.)

Creating GC_F (Garble**)**

1. Ginny first assigns each possible input with a label (random string). For instance, the input $g = 0$ is given the label W_g^0 and the input $g = 1$ is given the label W_g^1. Define W_e^0 and W_e^1 similarly, for the possible inputs of e.

2. Ginny hashes all combinations of g and e. For this particular example, there are 4 possible input combinations, giving us:

$$\begin{aligned} H_{00} &= \text{Hash}(W_g^0, W_e^0) \\ H_{01} &= \text{Hash}(W_g^0, W_e^1) \\ H_{10} &= \text{Hash}(W_g^1, W_e^0) \\ H_{11} &= \text{Hash}(W_g^1, W_e^1) \end{aligned}$$

   Note that Evan will also get access to Hash.

Sending GC_F to Evan (Encode**)**

1. Ginny encodes the outcomes $(g \wedge e)$ corresponding to each input, using the hashes from Garble as the secret keys. For our example, Ginny thus computes:

$$\begin{aligned} &\text{Encode}(H_{00}, 0), \text{ since } (0 \wedge 0) = 0 \\ &\text{Encode}(H_{01}, 0), \text{ since } (0 \wedge 1) = 0 \\ &\text{Encode}(H_{10}, 0), \text{ since } (1 \wedge 0) = 0 \\ &\text{Encode}(H_{11}, 1), \text{ since } (1 \wedge 1) = 1 \end{aligned}$$

2. Ginny now randomly permutes her encodings and sends all of them to Evan.

Recovering $F(x)$ (Eval**)**

1. Ginny sends Evan W_g^G, which corresponds to her actual input. If we want $(g \wedge e)$, Ginny also needs to send Evan W_e^E. However, Evan cannot give E directly to Ginny, and Ginny also should not leak W_e^{1-E} to Evan. This exchange of information can be achieved with an oblivious transfer protocol, described in Section 4.

2. Now, Evan has both W_g^G and W_e^E. He computes $H_{GE} = \text{Hash}(W_g^G, W_e^E)$, which is the secret key for one of the encodings. Evan can now try to decode each encoding using H_{GE}; when he is able to validly decode, he will get $(G \wedge E)$. Since he doesn't know what the random labels are (besides his own), he is unable to get any additional information from the encodings besides $(G \wedge E)$.

2.2 Composing Gates

Let's say we want to add another gate to the AND gate we garbled above, so that the output of $(g \wedge e)$ is an input to another AND gate, along with some other bit c.

In order to perform this composition of gates, we'll need a new set of symbols for the second gate. Let W_m^0 and W_m^1 be labels for the outcome of $(g \wedge e)$ being 0 and 1, respectively, and let W_c^0 and W_c^1 be labels for the bit c.

Then, when Ginny does the encoding phase for the $(g \wedge e)$ gate, instead of encoding 0 or 1, she can encode W_m^0 or W_m^1. She thus sends Evan a permuted version of:

$$\begin{aligned}
&\text{Encode}(H_{00}, W_m^0)\\
&\text{Encode}(H_{01}, W_m^0)\\
&\text{Encode}(H_{10}, W_m^0)\\
&\text{Encode}(H_{11}, W_m^1)
\end{aligned}$$

Ginny can construct the second gate in the exact same way as the first, except she will actually encode 0 or 1 rather than some label. Ginny can again use oblivious transfer to give Evan the appropriate W_c^C, and Evan can evaluate as we've described above.

The time needed to run the garbled circuit protocol grows about linearly in the number of gates we have, as the second garbled gate took just as much time to construct and evaluate as the first.

This can be generalized to other kinds of gates as well, like NOT, OR, and XOR. Being able to compose garbled versions of each of these gates gives us a fast way to evaluate any function F that can be expressed with these fundamental building blocks.

3 The Security of Yao

We now mention some key aspects of this scheme's security:

- We saw from the composition exercise above that an attacker could learn the size and layout of the circuit from GC_F. However, we see that due to the random labels and the encoding step (an attacker cannot learn anything from E_x, except size of input), the attacker cannot learn inputs to the gates, or what each specific gate does.

  To further obfuscate the circuit layout, it suffices to use the universal circuit, which actually takes in any circuit and its inputs and evaluates it. See [3] for more details. We can instead garble the universal circuit to hide the layout of the specific circuit we want to compute.

- The permutation step that Ginny runs after Encode is to ensure that Evan cannot use any sort of organization assumption (i.e. lexicographic ordering) to infer what exact input each encoding corresponds to. This protects us against a malicious Evan.

- Ginny may also be malicious! How does Evan know that she is actually sending valid labels? To lower the probability of Evan misevaluating the circuit, we can apply a cut and choose protocol. See [2] for more details.

- Ginny should never use the same garbled circuit twice. Evan could use this to his advantage to learn all of the labels. For instance, even in our simple AND gate example, if they run $(g \wedge e)$ twice, once with $e = 0$ and once with $e = 1$, Evan will learn both W_e^0 and W_e^1. Combining that with any W_g^G information he receives, he can get more information about the table of encodings that Ginny gives. In fact, if Ginny sends him both W_e^0 and W_e^1 over multiple computations over the same garbled circuit, Evan can reconstruct the entired garbled circuit!

4 Oblivious Transfer

Let E denote Evan's actual bit. Recall that in one step of the garbled circuit setup, Evan needs to receive W_e^E from Ginny, but: (1) Ginny should not know E, and (2) Evan should not know W_e^{1-E}. This can be achieved using "1-out-of-2 oblivious transfer," which relies on the *computational Diffie-Hellman assumption.*

4.1 Computational Diffie-Hellman

Let G be a group of order (i.e. size) odd prime p, and let g be a generator of G (i.e. each element of G can be written as g^a, where a is some natural number). We use $\text{negl}(n)$ to denote a function that is *negligible* in some parameter n, which means that the asymptotic behavior of the function is bounded above by $\frac{1}{\text{poly}(n)}$.

The computational Diffie-Hellman assumption formalizes the idea that knowing group elements g^x and g^y doesn't help you determine g^{xy}; we do not have any advantage over randomly guessing g^{xy}.

Definition 4.1. The *computational Diffie-Hellman assumption* states if A is an "efficient" (i.e. polynomial time) algorithm that takes in g^x, and g^y then:

$$\mathcal{A}(G, g, g^x, g^y) \neq g^{xy} \text{ with probability } 1 - \frac{1}{p} - \text{negl}(n)$$

Here, n is a security parameter.

4.2 Symmetric Key Encryption

The oblivious transfer protocol also uses a symmetric key encryption as a building block. We will define the encryption scheme and note some key properties.

Definition 4.2. Let $M \in \{0,1\}^m$ be a message, and let $k \in \{0,1\}^{m+n}$ be a secret key shared by the sender and the receiver. Write $k = x \| y$, where $\|$ denotes concatenation, x consists of the first m bits of k, and y consists of the last n bits of k.

- The encryption function is given by $E_k(M) = (x \oplus M) \| y$, where $\oplus$ is XOR.
- Let $e = e_1 \| e_2$ be an encrypted message, where e_1 consists of the first m bits and e_2 consists of the last n bits. The decryption function $D_k(e)$ outputs $\perp$ if $y \neq e_2$, and outputs $x \oplus e_1$ otherwise.

The key two properties of this scheme are highlighted below:

1. *Non-Committing:* This property says that there exists a polynomial time simulator Sim such that for all messages M and possible encryptions e, we output a key k (i.e. $\text{Sim}(e, M) = k$) such that $E_k(M) = e$.

   In other words, it is possible for a simulator to come up with a way to "explain" e as a valid encryption of M (by finding a valid k) in polynomial time.

2. *Robustness:* Let n be the security parameter. Let $\mathcal{S}$ be a random subset of all possible keys, and let $|\mathcal{S}|$ be polynomial in n. Robustness is satisfied if for all possible encryptions e:

   $$\Pr_{\mathcal{S}}[\text{at most one } k \in \mathcal{S} \text{ satisfies } D_k(e) \neq \perp] = 1 - \text{negl}(n)$$

   In other words, there is a low probability that an adversary generates an encryption e such that more than one accurate decryption of e can be found in polynomial time.

4.3 The Oblivious Transfer Scheme

Figure 1 depicts the diagram of the oblivious transfer protocol from [1]. The diagram summarizes all of the key steps, but we will elaborate on each step in more detail below.

We use a group G of order p, and we let $g \in G$ be a generator. We let M_0, M_1 denote the sender's messages corresponding to bits 0 and 1, respectively, and we let c denote the receiver's bit.

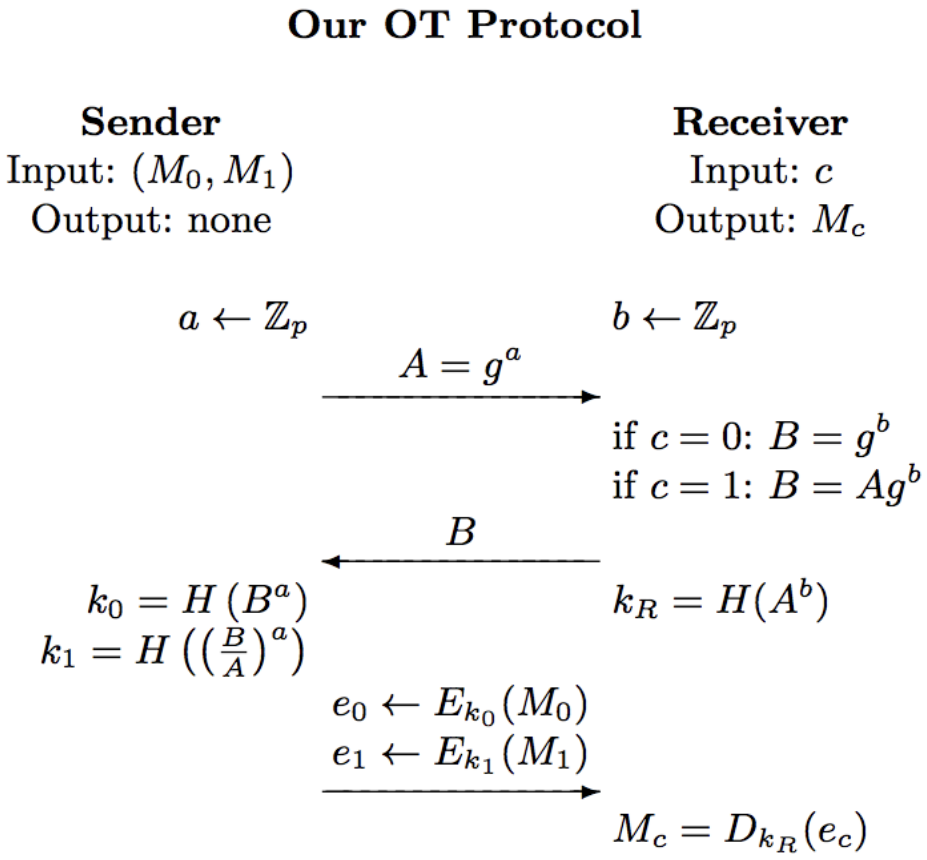

Figure 1: The oblivious transfer protocol of Chou and Orlandi [1]

1. The sender chooses $a \in \mathbb{Z}_p$ uniformly at random, and gives $A = g^a$ to the receiver.
2. The receiver chooses $b \in \mathbb{Z}_p$ uniformly at random, and gives $B = g^b \cdot A^c$ to the sender.
3. As the sender only knows B, it is equally likely from his point of view for c to be 0 and 1. This is because the bit b was also chosen uniformly at random. Now, the sender computes $k_0 = B^a$ and $k_1 = (\frac{B}{A})^a$, and sends both $E_{k_0}(M_0)$ and $E_{k_1}(M_1)$ to the receiver.
4. Now, the receiver computes $k_c = g^{ab} = A^b$, and tries to decode both $E_{k_0}(M_0)$ and $E_{k_1}(M_1)$ using k_c. The receiver will only be able to recover M_c:
   - If $c = 0$, then the sender computed:
   $$k_0 = B^a = (g^b \cdot A^0)^a = g^{ab} = k_c$$
   $$k_1 = (\frac{B}{A})^a = (\frac{g^b \cdot A^0}{g^a})^a = g^{(b-a)\cdot a} \neq k_c$$
   The receiver will get M_0 when applying D_{k_c} to $E_{k_0}(M_0)$, but will get $\perp$ when applying D_{k_c} to $E_{k_1}(M_1)$.
   - If $c = 1$, then the sender computed:
   $$k_0 = B^a = (g^b \cdot A^1)^a = g^{(a+b)a} \neq k_c$$
   $$k_1 = (\frac{B}{A})^a = (\frac{g^b \cdot A^1}{g^a})^a = g^{ba} = k_c$$
   The receiver will get M_1 when applying D_{k_c} to $E_{k_1}(M_1)$, but will get $\perp$ when applying D_{k_c} to $E_{k_0}(M_0)$.

4.4 Proof Sketches of Security

We sketch the proofs of the following two facts:

1. The sender does not learn the receiver's bit c.
   Proof: The only quantity related to c that the sender sees is $B = g^b \cdot A^c$. Since $c = 0$ or $c = 1$, the sender knows that either $B = g^{a+b}$ or $B = g^b$. However, since b is randomly chosen from $\mathbb{Z}_p$, the sender cannot distinguish g^{a+b} and g^b any better than by randomly guessing.
2. The receiver does not learn M_{1-c}.
   Proof: Say we have an adversarial receiver R^* that can see both keys $(B^a, (\frac{B}{A})^a)$ after just knowing G, g, and $A = g^a$. We represent this as:
   $$R^*(G, g, A) = (B^a, \left(\frac{B}{A}\right)^a)$$

We prove that the existence of R^* breaks computational Diffie-Hellman. We start with g^x and g^y, and try to learn g^{xy}. The adversary R^* can run the above operation on $A = g^x$, $A = g^y$, and $A = g^{x+y}$. (Note that we can get g^{x+y} easily from g^x and g^y by multiplying them.)

$$R^*(G, g, g^x) = (B^x, \left(\frac{B}{g^x}\right)^x)$$
$$R^*(G, g, g^y) = (B^y, \left(\frac{B}{g^y}\right)^y)$$

Using B^x and $(\frac{B}{g^x})^x$ we can learn $(B^x)/(\frac{B}{g^x})^x = (B^x)/(\frac{B^x}{g^{x^2}}) = g^{x^2}$. Similarly, we can learn g^{y^2} and $g^{(x+y)^2}$. Using these quantities, we can learn g^{2xy}:

$$\frac{g^{(x+y)^2}}{g^{x^2} \cdot g^{y^2}} = \frac{g^{x^2+2xy+y^2}}{g^{x^2+y^2}} = g^{2xy}$$

To obtain g^{xy} from g^{2xy}, we can raise g^{2xy} to the $\frac{p+1}{2}$ power. (Recall that p is an odd prime, so $\frac{p+1}{2}$ is an integer.) Then, we obtain

$$g^{(p+1)xy} = (g^{xy})^{(p+1)}$$

Since G has order p, for any $g' \in G$, we have $(g')^p = \text{id}$ (by Fermat's Little Theorem). This allows us to conclude:

$$(g^{xy})^{p+1} = (g^{xy})^p \cdot g^{xy} = \text{id} \cdot g^{xy} = g^{xy}$$

By assuming existence of R^*, we have broken computation Diffie-Hellman.

References

[1] Chou, Tung, and Claudio Orlandi. "The simplest protocol for oblivious transfer." International Conference on Cryptology and Information Security in Latin America. Springer, Cham, 2015.

[2] Crépeau, Claude. "Cut-and-choose protocol." Encyclopedia of Cryptography and Security. Springer, Boston, MA, 2011. 290-291.

[3] Valiant, Leslie G. "Universal circuits (preliminary report)." Proceedings of the eighth annual ACM symposium on Theory of computing. ACM, 1976.

[4] Yakoubov, Sophia. "A Gentle Introduction to Yao's Garbled Circuits (2017)."