Garbled Circuits

David Heath

Fall 2025

So far, we have seen general-purpose MPC protocols that are based on a paradigm of additive (in particular, XOR) secret sharing. Indeed, last time we saw how to use this paradigm to achieve general-purpose semi-honest MPC: We have a single protocol (the GMW protocol) that works for any number of parties p and any Boolean circuit C.

However, as we have discussed, this protocol has several distinct shortcomings. The shortcoming we will address today is the *high round complexity* inherent to the protocols we have seen so far. In particular, recall that all protocols we have seen so far incur a number of protocol rounds propotional to the so-called *multiplicative depth* of the circuit C. It's natural to ask: can we get protocols with better round complexity?

Indeed, we can. In fact, we will see that it is possible to build secure protocols for any computable function that run in a *constant* number of protocol rounds. The main trick for achieving such protocols is called the Garbled Circuit (GC) technique, first described by Andrew Yao [Yao86].

Very roughly, GC allows a party to construct an *encrypted version* of a particular program. This encrypted program can be evaluated by some other party exactly once, with the property that they are able to successfully "decrypt an execution" of the program, but while learning nothing about values internal to the program execution. Our goal today will be to understand this rough description in more detail.

It's perhaps useful to understand GC as a *distinct paradigm* of how to achieve MPC; it is not superior to the secret-share-based techniques we have seen, but rather it is *incomparable*. While we will achieve better round complexity, we will do so at the cost of increased communication.

Preliminaries: Symmetric Key Encryption

The idea that GC is a mechanism for *encrypting a program* is relatively literal. Indeed, we will need a symmetric-key encryption scheme to build up the GC technique.

Definition 1 (Symmetric Key Encryption with CPA$ Security)**.** *A **symmetric-key encryption scheme** is a triple of algorithms* $(\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec})$ *with the following interface:*

- $\texttt{KeyGen}$ *takes as input a security parameter* λ*; it outputs a key* k *drawn a set* $\mathcal{K}$ *(the* key space*).*
- $\texttt{Enc}$ *takes as input a key* $k \in \mathcal{K}$ *and a message* $m \in \mathcal{M}$*, where* $\mathcal{M}$ *is a set of allowed messages (the* message space*); it outputs a ciphertext* $c \in \mathcal{C}$ *where* $\mathcal{C}$ *is a set of ciphertexts (the* ciphertext space*).*
- $\texttt{Dec}$ *takes as input a key* $k \in \mathcal{K}$ *and a ciphertext* $c \in \mathcal{C}$*; it outputs a message* $m \in \mathcal{M}$*.*

The scheme is (perfectly) ***correct*** *if the following holds for all keys* $k \in \mathcal{K}$ *and all messages* $m \in \mathcal{M}$*:*

$$\texttt{Dec}(k, \texttt{Enc}(k, m)) = m$$

The scheme has ***pseudorandom ciphertexts under chosen plaintext attacks*** *(denoted CPA$) if the following two programs are computationally indistinguishable (in* λ*):*

```
Real :
  private k ← KeyGen(λ)
  public lookup(m ∈ M) :
    return Enc(k, m)
```

$\stackrel{c}{=}$

```
Ideal :
  public lookup(m ∈ M) :
    c ∈$ C
    return c
```

In short, such an encryption scheme produces ciphertexts that look random to an adversary who does not know the encryption key.

Warm-Up: Encrypting an AND Gate

Let's consider a single AND gate $z \leftarrow x \cdot y$, and let's invent a contrived scenario:

> Alice has two secret messages m_0 and m_1. She wishes to associate with the two input wires x and y encryption keys. In particular, each wire will be associated with *two* encryption keys, one corresponding to logical zero, and one to logical one. Let's call these keys $K_x^0, K_x^1, K_y^0, K_y^1$. Further suppose that Bob somehow magically obtains one key per input wire. Suppose also that Bob also knows x and y. Alice wants to arrange that if Bob obtains both one keys K_x^1, K_y^1, then Bob can learn m_1, but not m_0. In all other cases, she wishes Bob to learn m_0, but not m_1.

To more specific, let's suppose we have a symmetric-key encryption scheme (Definition 1), and let's suppose that the keys $K_x^0, K_x^1, K_y^0, K_y^1$ are all drawn by calling `KeyGen`:

$$K_x^0 \leftarrow \texttt{KeyGen}(\lambda) \qquad K_x^1 \leftarrow \texttt{KeyGen}(\lambda) \qquad K_y^0 \leftarrow \texttt{KeyGen}(\lambda) \qquad K_y^1 \leftarrow \texttt{KeyGen}(\lambda)$$

Let's assume Alice knows all four keys (indeed, later she will choose them herself).

Alice can arrange that Bob learns the appropriate message m_0, m_1 by sending to Bob the following four "double" encryptions:

$$\begin{aligned}
&\texttt{Enc}(K_x^0, \texttt{Enc}(K_y^0, m_0)) \\
&\texttt{Enc}(K_x^0, \texttt{Enc}(K_y^1, m_0)) \\
&\texttt{Enc}(K_x^1, \texttt{Enc}(K_y^0, m_0)) \\
&\texttt{Enc}(K_x^1, \texttt{Enc}(K_y^1, m_1))
\end{aligned}$$

In a sense, the above four encryptions constitute an **encrypted truth table** corresponding to the logic of an AND gate.

Notice that sending this information is indeed sufficient to achieve our contrived goal: Bob, who holds one key K_x^* and one key K_y^* can correctly decrypt the row corresponding to a secret he is supposed to hold. (Let's ignore for now the problem of how Bob *knows* which row to decrypt; we will discuss that later.) Morever, Bob *cannot* decrypt any of the other three rows. For instance, suppose Bob holds keys K_x^1, K_y^1. Then to Bob, the encrypted truth table looks like this (in a sense that can be formalized as a simulator):

$$\begin{aligned}
&\$ \\
&\$ \\
&\texttt{Enc}(K_x^1, \$) \\
&\texttt{Enc}(K_x^1, \texttt{Enc}(K_y^1, m_1))
\end{aligned}$$

Here, \$ is an informal shorthand meaning "looks like a uniform ciphertext". Similarly, if Bob holds *any* pair of keys, we can simulate what he sees.

Exercise 1. *Explain why the third row above looks like* $\texttt{Enc}(K_x^1, \$)$*, and not just* \$*.*

Exercise 2. *Formalize this contrived scenario and simulate Bob's view.*

Encrypting Circuits

Now, let's generalize our above contrived scenario. In particular, Alice still has messages m_0 and m_1, but now she would like to encrypt those messages according to evaluation not just of an AND gate, but rather

an entire Boolean circuit $C : \{0,1\}^n \to \{0,1\}$. As above, we will associate each of C's input wires with two keys. Alice wishes for Bob to be able to decrypt m_1 if Bob holds an input x—and corresponding keys—s.t. $C(x) = 1$; else, Bob should be able to decrypt m_0.

The high-level idea is a straightforward extension of the above observation about AND gates: Alice will set up encrypted truth tables corresponding to each gate in C. For each such gate, she will arrange that evaluating (decrypting) that gate allows Bob to obtain some other key, which will allow him to decrypt subsequent gates, until he ultimately obtains a key on C's output wire, which he can use to decrypt eith m_0 or m_1.

In a bit more detail, Alice will first associate with every wire w in the circuit two fresh wire-specific keys:

$$K_w^0 \leftarrow \texttt{KeyGen}(\lambda) \qquad K_w^1 \leftarrow \texttt{KeyGen}(\lambda)$$

Then—using the same high-level idea as before—Alice can gate-by-gate encrypt the truth table of each gate, choosing the gate-encrypted to be *the keys associated with the output of that gate.* For example, consider some AND gate $z \leftarrow x \cdot y$; Alice can send the following to Bob:

$$\begin{aligned}
&\texttt{Enc}(K_x^0, \texttt{Enc}(K_y^0, K_z^0)) \\
&\texttt{Enc}(K_x^0, \texttt{Enc}(K_y^1, K_z^0)) \\
&\texttt{Enc}(K_x^1, \texttt{Enc}(K_y^0, K_z^0)) \\
&\texttt{Enc}(K_x^1, \texttt{Enc}(K_y^1, K_z^1))
\end{aligned}$$

Similarly Alice can encrypt the logic of *any* gate-type, e.g. XOR or OR, and send the encryption of *each* gate to Bob. The collection of all such encryptions is called a **garbled circuit**. Now, once Bob obtains keys on the circuit's input wires, he can one-by-one decrypt each gate and ultimately obtain keys on the circuit's output wire `out`. Finally, Alice sends:

$$\texttt{Enc}(K_{\texttt{out}}^0, m_0)) \qquad \texttt{Enc}(K_{\texttt{out}}^1, m_1))$$

This allows Bob to decrypt the single message $m_{C(x)}$.

Remark 1 (Non-interactivity)**.** *Notice crucially that Bob's evaluation of the encrypted circuit is non-interactive. Once Bob obtains input keys and the encrypted circuit, he evaluates the encrypted circuit without Alice's help. This non-interactivity is ultimately how we will obtain constant round MPC protocols.*

Point and Permute

We are now almost ready to give a full construction of a so-called *garbling scheme* [BHR12]—a mechanism for encrypting computations. However, so far we have sketched only a so-called *privacy-free* garbling scheme; namely, we have assumed that Bob knows the input to the circuit C. To achieve MPC protocols from this primitive, it will be important that Bob learn *nothing* beyond the output of the circuit $m_{C(x)}$.

In particular, Bob knows the circuit C. He holds a garbled circuit and input wire keys corresponding to some input x, but he does not know x, and *he should not learn* x.

Stripping Bob of the input x introduces two problems, one in terms of correctness, and one in terms of security:

- **Correctness.** For each garbled gate, Bob is faced with four random-looking encryptions. If he does not know which input keys he holds, how does he know which of the four rows to decrypt?
- **Security.** If Bob *does* learn which row i to decrypt, since the row ID i leaks the gate inputs.

These two problems can be simultaneously resolved by a standard trick called *point and permute* [BMR90]. The idea is that each wire key will now include (1) an encryption key K_w^* and (2) a single so-called *color*

bit. This color bit is random, but Alice will ensure that each wire's pair of keys have differing colors. In particular, for each wire w, she now samples the wire keys as follows:

$$
\begin{aligned}
&c \in_{\$} \{0,1\} \\
&k^0 \leftarrow \texttt{KeyGen}(\lambda) \\
&k^1 \leftarrow \texttt{KeyGen}(\lambda) \\
&K_w^0 = (k^0, c) \\
&K_w^1 = (k^1, \neg c)
\end{aligned}
$$

Now, to garble a particular gate—say an AND gate $z \leftarrow x \cdot y$—she does so in the same manner as before, except that she *permutes* the order of rows according to colors on the input wires. Bob uses the colors on the keys he sees to decide which row to decrypt. This gives correctness (Bob knows which row to decrypt) and security (the rows are randomly permuted).

Formally, here's Alice's procedure for garbling an AND gate. Recall that each key K_w^* has two parts, a cryptographic key and a color bit:

$$
\begin{aligned}
&\texttt{GarbleAND}((K_x^0, K_x^1, K_y^0, K_y^1, K_z^0, K_z^1)) : \\
&\quad (k_x^0, c) \leftarrow K_x^0 \\
&\quad (k_x^1, \cdot) \leftarrow K_x^1 \\
&\quad (k_y^0, d) \leftarrow K_y^0 \\
&\quad (k_y^1, \cdot) \leftarrow K_y^1 \\
&\quad r_{00} \leftarrow \texttt{Enc}(k_x^c, \texttt{Enc}(k_y^d, K_z^{c \cdot d})) \\
&\quad r_{01} \leftarrow \texttt{Enc}(k_x^c, \texttt{Enc}(k_y^{\neg d}, K_z^{c \cdot \neg d})) \\
&\quad r_{10} \leftarrow \texttt{Enc}(k_x^{\neg c}, \texttt{Enc}(k_y^d, K_z^{\neg c \cdot d})) \\
&\quad r_{11} \leftarrow \texttt{Enc}(k_x^{\neg c}, \texttt{Enc}(k_y^{\neg d}, K_z^{\neg c \cdot \neg d})) \\
&\quad \texttt{return } (r_{00}, r_{01}, r_{10}, r_{11})
\end{aligned}
$$

Alice sends the four rows r_{ij} to Bob.

Exercise 3. *Try all four combinations of colors c, d and ensure you understand what happens to the garbled truth table.*

During evaluation, Bob uses the colors on his keys to decide which row to decrypt. Below, we write K_x to denote whichever key Bob happens to hold on wire x; i.e. $K_x \in \{K_x^0, K_x^1\}$. Again, Bob doesn't know—and should not learn—which of the two keys he holds.

$$
\begin{aligned}
&\texttt{EvalAND}((K_x, K_y, r_{00}, r_{01}, r_{10}, r_{11})) : \\
&\quad (k_x, c) \leftarrow K_x \\
&\quad (k_y, d) \leftarrow K_y \\
&\quad K_z \leftarrow \texttt{Dec}(k_y, \texttt{Dec}(k_x, r_{cd}))
\end{aligned}
$$

Exercise 4. *Check you see how the above procedures generalize to* any *fan-in two gate, not just AND gates. Think how to generalize beyond two-input gates. What's the problem with this generalization?*

Exercise 5. *Check that the above two procedures are* ***correct****. Namely, if Alice garbles an AND gate, then Bob evaluates with matching input keys, he obtains an appropriate output key.*

A Garbling Scheme for Boolean Circuits

Now, we can sketch a *garbling scheme* for Boolean circuits. In particular, Alice can garble an arbitrary Boolean circuit as described above: she samples a pair of keys for each wire in the circuit, and then for each gate in the circuit, she garbles a gate. When Bob, the evaluator, obtains input keys and the garbling of each gate, he can decrypt and ultimately obtain an output.

This scheme has an important security property that we can call *privacy*:

Definition 2 (Garbled Circuit Privacy)**.** *There exists a simulator* `Sim` *that simulates Bob's view of a garbled circuit and garbled input. Formally, let* $C : \{0,1\}^n \to \{0,1\}^m$ *be an arbitrary Boolean circuit. Let* $x \in \{0,1\}^n$ *be a cleartext input, and let* $y = C(x)$ *be the corresponding cleartext output. Suppose Alice garbles* C*, yielding garbled circuit (i.e., the collection of gate encryptions)* $\tilde{C}$*, as well as pairs of keys on input wires (let's call this collection of pairs of keys* e *for* encoding *string). Now, suppose Bob obtains* $\tilde{x}$*, the input keys corresponding to* x*. Namely, let's write* $\tilde{x} \leftarrow \texttt{Encode}(e, x)$*; i.e.* `Encode` *is simply a procedure that selects appropriate input keys corresponding to* x*. The following indistinguishability holds:*

$$\left\{ (C, \tilde{C}, \tilde{x}) \quad \textit{where} \quad (\tilde{C}, e) \leftarrow \texttt{Garble}(\lambda, C), \tilde{x} \leftarrow \texttt{Encode}(e, x) \right\} \stackrel{c}{=} \{ \texttt{Sim}(\lambda, C, y) \}$$

Roughly, the simulator just simulates each garbled gate by (1) randomly sampling *one* key per wire, (2) arranging a randomly permuted "garbled truth table", where exactly one row is a valid encryption, and the others are random (or an encryption of random, recall our warm-up), then (3) setting up an encryption of the appropriate output y under the randomly chosen keys.

Exercise 6. *Proving indistinguishability for the above simulator works is non-trivial. Try thinking through how such an argument might go. You can see a detailed argument here [LP09], though they unfortunately do not use the important point and permute optimization.*

2PC in Constant Rounds

Given our garbling technique, achieving semi-honest secure 2PC is straightforward. Here's the protocol:

- Alice and Bob agree on a Boolean circuit C.
- Alice garbles the circuit, yielding garbled circuit $\tilde{C}$ and input keys e.
- Alice and Bob run Oblivious Transfer to allow Bob to select input keys corresponding to his input y.
- Alice sends $\tilde{C}$, as well as input keys corresponding to her input x.
- Bob evaluates the garbled circuit and obtains $C(x, y)$; he sends the output to Alice.

Note crucially that this runs in a constant number of rounds!

Next Time

We now have a 2PC protocol for arbitrary Boolean circuits. However, unlike secret-share-based MPC, the 2PC protocol does no *immediately* generalize. Still, we can upgrade GC to work for more than two parties, and next time we will see how that is done.

References

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols (extended abstract). In *22nd ACM STOC*, pages 503–513. ACM Press, May 1990.

[LP09] Yehuda Lindell and Benny Pinkas. A proof of security of Yao's protocol for two-party computation. *Journal of Cryptology*, 22(2):161–188, April 2009.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *27th FOCS*, pages 162–167. IEEE Computer Society Press, October 1986.