Multiparty Garbled Circuits

David Heath

Fall 2025

Last time, we introduced the Garbled Circuit (GC) technique [Yao86]. Recall that GC is interesting because it enables secure 2PC protocols that run in a small constant number of rounds. In particular, last time we saw a semi-honest 2PC protocol that proceeds as follows:

- The parties agree on a Boolean circuit C expressing their desired computation.
- Alice plays the role of the so-called *garbler*. She "garbles" C, which involves assigning two encryption keys (one per logical value) to each of the wires in C, then using those keys to encrypt the logic of each of the gates in C. The result is a collection of gate encryptions—the garbled circuit $\tilde{C}$.
- Alice and Bob interact such that Bob learns one key per input wire. This is achieved via calls to oblivious transfer.
- Alice sends the garbled circuit $\tilde{C}$ to Bob, who is playing the role of the so-called *evaluator*.
- Bob *evaluates* the garbled circuit. He walks through C in a topological order, and at each gate he uses keys on gate input wires to decrypt part of the gate encryption in $\tilde{C}$, yielding a key on the gate output wire. He does this for every gate, ultimately obtaining one key per circuit output wire. Alice can straightforwardly arrange that Bob can use these output keys to decrypt the cleartext output.
- To complete the protocol, Bob can send the output to Alice.

While this works, the technique is somewhat fundamentally a two-party one. We might wish generalize to constant-round p-party secure computation, where even if $p-1$ parties collude, the remaining party still has a guarantee of privacy. Recall that in the interactive MPC setting, the classic GMW protocol *naturally* scales from 2PC to MPC. It is not obvious how to generalize the above protocol to a p-party one.

Today, we will investigate how to achieve a p-party MPC protocol with semi-honest security. The protocol we will study was proposed by Beaver, Micali, and Rogaway [BMR90]. Let n denote the number of gates in the computed circuit C. Their protocol achieves semi-honest p-party computation in constant rounds and within $O(n \cdot p^2 \cdot \lambda)$ bits of communication, where our security parameter λ denotes the length of encryption keys. This is remarkably good for the considered setting, and only very recently have any substantial improvements been made (in the semi-honest setting).

An Attempt that Does Not Work

Consider the following (incorrect) attempt at generalizing the above protocol to a p-party protocol:

- One party serves as the garbler. That party garbles the circuit and sends the garbled circuit to each other party.
- Parties use OT (perhaps in some complicated fashion) to select input keys.
- Each party evaluates the garbled circuit locally and obtains output.

The above intuition can be formalized into a protocol that is *correct* (i.e., the parties obtain the correct output), but not one that is *secure* in our considered setting.

The problem is that if Alice colludes with even one of the evaluators, she will immediately learn all party inputs, breaking security. In particular, each evaluator holds wire input keys. While to each evaluator each input key just looks random, Alice knows the association between keys and logical values, so she can immediately decrypt the input.

While this attempt does not work, it is instructive: if we assume that Alice does not collude with any of the evaluators, then we *could* achieve a provably-secure protocol. This is to our assumption earlier in this course of an *untrusted dealer* for constructing Beaver multiplication triples. In this case, the untrusted dealer prepares a garbled circuit and distributes it amongst the parties.

An Instructive (But Ultimately Wrong) Intuition

Based on the above observation, we have a logical way to proceed: implement the untrusted dealer Alice via a cryptographic protocol! Recall that this is exactly what we did in the GMW protocol – we replaced our untrusted dealer's multiplication triples by calls to OT. We can perform a similar transformation here.

Namely, observe that the garbler's behavior is simply a computation, and we already have a protocol for securely executing *arbitrary* computations – the GMW protocol! So as a sketch of how to proceed, we can simply use the GMW protocol to implement the garbler. Then parties can somehow use OTs to distribute garbled input keys amongst themselves and locally compute the output. This almost works. It is correct, and it is secure.

The sketch even achieves a relatively low round complexity, even though it makes use of the GMW protocol, which has high round complexity. To see this, it's crucial to observe that our approach to garbled circuits makes it possible for the garbler to encrypt all circuit gates *in parallel.* This means that in the GMW protocol, we can also compute all gate encryptions in parallel, meaning that the computation completes after a small number of rounds.

While this sketch[1] works, it does not achieve our desired complexity. Recall that our garbling procedure from last time used an encryption $(\texttt{KeyGen}, \texttt{Enc}, \texttt{Dec})$ to encrypt each of the gates. Thus, our sketch must run the procedure `Enc` *inside the GMW protocol.* Therefore, the communication and round complexity of our sketched protocol *depends on the complexity of* `Enc` *expressed as a Boolean circuit.* This very substantially increases[2] the cost of the protocol, both in terms of communication and rounds. Formally, the best we can say, without knowing more about our encryption scheme, is that our complexities scale with some unspecified $O(\texttt{poly}(\lambda))$ factor.

Remark 1 (Non-Black-Box Cryptography)**.** *Our sketch leverages a technique that we have not yet seen in this course –* ***non-black-box cryptography****. Notice that to actually run the protocol, the parties need not just a symmetric-key encryption scheme, they also need* the code for `Enc`*, written as a circuit. Indeed, they need* `Enc`*'s code so that they can use it as a parameter of the GMW protocol. In all of our prior uses of cryptography, it was sufficient to have* black-box *access to cryptographic algorithms, where it mattered only what properties those algorithms gave us, not how they were implemented. The use/non-use of non-black-box cryptography is a qualitative metric by which we can evaluate a protocol. Informally, non-black-box cryptography is often synonymous with "extremely inefficient".*

The BMR Protocol

Let's upgrade our above sketch to achieve our efficiency goal. We will achieve p-party computation from OT and any (black-box) *pseudorandom generator.*

[1]Interestingly, even experts in the area are sometimes convinced that this sketch *is* the constant round protocol as described by BMR. Unfortunately, the approach is not quite this simple, and additional insights are needed.

[2]For reference, the Advanced Encryption Standard (AES, a typical construction used to implement symmetric-key encryption) circuit includes 6400 AND gates, and it's multiplicative depth is hundreds of gates long. Recall that GMW requires $\approx p^2$ OTs per AND gate, so running AES inside GMW requires $\approx 6400p^2$ OTs, and we must run AES multiple times to garble each gate in the target circuit. To put it lightly, this is impractical.

Definition 1 (Pseudorandom Generator (PRG))**.** *A **PRG** is a deterministic algorithm G. For simplicity, we will assume $G : \{0,1\}^\lambda \to \{0,1\}^m$ takes as input λ bits and outputs any number m of desired bits. Security of G is defined by computational indistinguishability of the following two programs:*

$$\boxed{\begin{array}{l}\texttt{Real}() : \\ \quad s \in_\$ \{0,1\}^\lambda \\ \quad \texttt{return}\ G(s)\end{array}} \qquad \overset{c}{=} \qquad \boxed{\begin{array}{l}\texttt{Ideal}() : \\ \quad r \in_\$ \{0,1\}^m \\ \quad \texttt{return}\ r\end{array}}$$

In 2PC garbling, recall that the garbler associates with each wire two keys. Those keys are used to encrypt logical truth tables. The approach here will be similar, with the following crucial insight:

> Each party will play the role of garbler. Each party associates two keys with each wire. Gate truth tables are encrypted with *each party's keys*. Each party will also play the role of evaluator. For each circuit wire, each evaluator will see one key per garbler (i.e., in total the evaluator sees p keys per wire).

To make this work, the parties will ensure that for each wire—unlike in the 2PC garbling case—each garbler *does not know the association between his keys and logical values.* We can break this association by randomly swapping each wire's keys, which can be done efficiently inside the GMW protocol. This random swap is achieved by associating with each wire a uniform bit—a color bit, as discussed last time—and swapping keys according to that bit.

In more detail, let us consider some wire x. Each garbler P_i associates two uniform keys $X_i^0, X_i^1 \in_\$ \{0,1\}^\lambda$ with wire x. Each party then secret shares both keys, such that they can be used in the GMW protocol. Thus, the parties jointly hold the following secret sharings:

$$[X_0^0], ..., [X_{p-1}^0] \qquad [X_0^1], ..., [X_{p-1}^1]$$

It will be convenient to think of the following length $p\lambda$ strings, which are simply the concatenation of party keys:

$$X^0 \triangleq X_0^0 || ... || X_{p-1}^0 \qquad X^1 \triangleq X_0^1 || ... || X_{p-1}^1$$

Note that parties implicitly hold $[X^0]$ and $[X^1]$. The parties jointly sample a uniformly sampled color bit $[\alpha]$ where $\alpha \in_\$ \{0,1\}$ (recall, it is easy to sample a secret-shared bit by having each party sample its own share).

Invariant 1. *Just like our previous constructions of MPC protocols, we will achieve a correct and secure protocol by ensuring an invariant holds for each circuit wire x. Namely, at runtime, each evaluator will learn $X^{x \oplus \alpha}$ (and not $X^{x \oplus \alpha \oplus 1}$). Notice that this means that even though a particular party knows part of both keys, they do not know α, and hence they do not learn the cleartext value x.*

Now, consider an AND gate $z \leftarrow x \cdot y$ (other gate types are handled in an essentially identical manner). The parties would like to jointly garble this AND gate. Let X^0, X^1 denote the keys on X, Y^0, Y^1 the keys on y, and Z^0, Z^1 the keys on y. Let x have color bit α, y have color bit β, and z have color bit γ.

Later on, when an evaluator actually runs the garbled circuit, by Invariant 1 we will ensure that they hold $X^{x \oplus \alpha}$ and $Y^{y \oplus \beta}$. Our goal is to arrange that they will therefore be able to decrypt $Z^{z \oplus \gamma}$, where $z = x \cdot y$.

To explain how the parties garble this gate, it will be helpful to define the following syntax. Let $S = S_0, ..., S_{p-1}$ denote a length $p\lambda$ string consisting of per-garbler keys. We define the following shorthand:

$$G(S) \triangleq G(S_0) \oplus ... \oplus G(S_{p-1})$$

I.e., we define $G(S)$ denotes calls G on each of the keys in S, then XORing the results. Here is the four-row garbled gate we would like the evaluator to obtain (note that we are not yet discussing how this garbled gate is constructed, just what it looks like):

$$\begin{aligned}
&G(X^0) \oplus G(Y^0) \oplus Z^{\alpha\cdot\beta\oplus\gamma}\\
&G(X^0) \oplus G(Y^1) \oplus Z^{\alpha\cdot\neg\beta\oplus\gamma}\\
&G(X^1) \oplus G(Y^0) \oplus Z^{\neg\alpha\cdot\beta\oplus\gamma}\\
&G(X^1) \oplus G(Y^1) \oplus Z^{\neg\alpha\cdot\neg\beta\oplus\gamma}
\end{aligned}$$

While the above is notationally heavy, it carries the same main idea as what we saw last time: encrypt an output key using all four possible combinations of input keys. Notice that the encrypted values Z^* are permuted according to color bits.

Now, to complete the construction of the protocol, we observe that the above garbled truth table can be efficiently constructed using the GMW protocol, while making only black-box calls to G. First, observe it is easy to compute secret shares $[G(X^0)]$ (resp. $[G(X^1)], G(Y^0), G(Y,1)]$): Each garbler i locally computes $G(X_i^0)$; since $G(X^0)$ is defined as the sum of such values, this implicitly defines a secret share.

Thus, the only remaining challenge is to compute secret shares of the encrypted values Z^*. But this is also easily achieved, since each garbler i knows Z_i^0, Z_i^1 in cleartext. For example let's focus on handling the first row, to compute $[Z^{\alpha\cdot\beta\oplus\gamma}]$, the parties first use GMW to compute $[\alpha \cdot \beta \oplus \gamma]$. Let's call the resulting secret-shared bit ρ; the parties hold $[\rho]$. They now wish to compute $[Z^\rho]$, which can be achieved by the following Boolean computation:

$$(Z^0 \oplus Z^1) \cdot \rho \oplus Z^0 = \begin{cases} Z^0 & \text{if } \rho = 0 \\ Z^1 & \text{otherwise} \end{cases} = Z^\rho$$

Exercise 1. *As written, computing Z^ρ inside GMW requires $p\lambda$ AND gates, simply because Z^0, Z^1 each have length $p\lambda$. Thus, naïvely this would require $O(p^3\lambda)$ calls to OT. However, the above computation has a special form: it is a vector-scalar multiplication of a bit ρ with a vector $(Z^0 \oplus Z^1)$. Investigate how to use this special form to reduce the number of OTs to only $O(p^2)$.*

Thus, after the garblers use GMW to compute $[Z^\rho]$, they garblers hold $[G(X^0)], [G(Y^0)]$, and $[Z^\rho]$; hence, they can locally compute a secret-sharing of the garbled row $[G(X^0) \oplus G(Y^0) \oplus Z^\rho]$. By in parallel computing all four rows for each of the gates, the parties can indeed construct the required garbled circuit in a secret-shared fashion in a constant number of rounds. From here, the parties can reveal the garbled circuit to themselves by exchanging shares.

A remaining challenge is in how to obtain keys on input wires. This is also not hard. Consider some wire x belonging to the input of some particular party P_i. Let that wire have color bit α. The parties (1) compute $[\alpha]$ inside GMW, then (2) reveal α to (only) party P_i by sending him their shares. Note that it is safe for P_i to know the color of this wire, since it belongs to his input. P_i computes $x \oplus \alpha$, secret shares it, and then the parties again use GMW evaluation to compute $[X^{x\oplus\alpha}]$ as already described. They reveal this to themselves to get the input key. By handling all inputs in parallel, the parties are ready to evaluate.

Finally, the parties must decrypt outputs. This is straightforward: the parties just exchange shares of color bits on output wires. This is safe because the parties are supposed to learn the cleartext value on output wires.

This completes the protocol. Notice that because all wires/gates are handled in parallel, the protocol completes in a constant number of rounds. The main technical insight was to allow each party to encrypt, using their own key pair, each row of each garbled truth table. Thus, each row is encrypted p times. Compared to our earlier attempt, this increased the size of the garbled circuit by factor p, but it allowed us to achieve security while using only black-box cryptography.

Next Time

All general-purpose protocols we have seen require extensive use of OT. For instance, the protocol today involved $O(p^2)$ OTs per gate. Unfortunately, based on what we have currently studied, each of these OTs involves a call to a public key encryption scheme, which is very expensive. Next time, we will explore ways to greatly improve the efficiency of OT.

References

[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.

[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.