# Two-Party Computation via Yao's Garbled Circuits

Noah Stephens-Davidowitz

June 9, 2023

## Recap

In the previous few lectures, we (1) saw the definition of secure two-party computation (in the honest-but-curious setting); (2) defined and constructed oblivious transfer; and (3) saw the GMW protocol, which uses OT to build arbitrary secure two-party computation.

The GMW protocol is great, but for the sake of argument, let's find some reasons to be unsatisfied with it. In particular, recall that the GMW protocol needed to perform a (one-out-of-four) oblivious transfer for every *gate* in the circuit that we wish to compute. Of course, the kinds of programs that we like to run usually have at least thousands of gates, and often have billions of gates. (The number of gates is essentially the number of simple bit operations necessary to compute our function.) Running OT this many times is quite impractical. E.g., the oblivious transfer protocols that we have seen require us to perform group exponentiation on groups with size $\approx 2^{1000}$ in order to be secure in practice. So, each oblivious transfer would itself require roughly billions of clock cycles, and kilobytes of communication. This makes running such a protocol with billions of gates completely impractical. (You can be a little smarter and group the oblivious transfers together, but you will still get quite an inefficient protocol. You can also make things significantly more efficient in practice by recalling that XOR and NOT gates are essentially free for GMW. So, you can try to construct a circuit for your function that consists of mostly XOR and NOT gates.)

Furthermore, these OT protocols must be run *in series*. In particular, Bob must know the results of the OT corresponding to the parent gates before running the OT corresponding to the child gate. This means that the number of individual messages sent by Alice and Bob (or the number of *rounds* of the protocol) must be quite large. (If you're careful, you can make the number of rounds proportional to the *depth* of the circuit, by performing all of the OTs corresponding to a single *level* of the circuit in parallel. But, you cannot run the OT for a parent gate in parallel with its child gate.)

So, having found some reasons to complain about GMW, we are now ready to see another construction of secure two-party computation, which will not have these issues. (Of course, we really just want to see another construction because secure two-party computation is interesting, and it's nice to see two very different ways to do it. Plus, the construction that we're going to see is quite strange and beautiful.)

## 1 Yao's garbled circuits

The construction that we will see today is due to Andrew Yao, and is referred to as Yao's garbled circuit protocol. (Yao famously never actually published this construction. Instead, he published a simpler construction for a special case of two-party computation [Yao86], but in his conference talk, he reportedly described the more general protocol. This is not a great way to do research, but Yao is cool enough that he can get away with it. So, we all now credit the paper [Yao86] with defining garbled circuits, though it absolutely did not. The phrase "garbled circuit" was coined in [BMR90].)

Garbled circuits have the following behavior. There is a PPT algorithm called $\mathsf{Garble}$, which takes as input a circuit $C$ (and the security parameter) with $\ell$ input gates and outputs two things: a "garbled" version of the circuit $\widetilde{C}$ and "garbled" versions of the possible input bits $k_1^{(0)}, k_1^{(1)}, \ldots, k_\ell^{(0)}, k_\ell^{(1)}$. We think of $k_i^{(b)}$ as representing an input whose $i$th bit is $b$. There is also a PPT algorithm $\mathsf{Eval}$, which takes as input $\widetilde{C}$ together with $k_1^{(x_1)}, \ldots, k_\ell^{(x_\ell)}$ for some input bits $x = (x_1, \ldots, x_\ell) \in \{0,1\}^\ell$, and outputs $C(x)$ (i.e., the circuit $C$ evaluated on input $x$). In other words, given a "garbling" $\widetilde{C}$ of the circuit $C$ and a "garbling" $(k_i^{(x_i)})_{i=1}^\ell$ of the input $x$, it is possible to compute the output $C(x)$ efficiently.

The security property that we want is that "the garbling $\widetilde{C}$ together with $k_1^{(x_1)}, \ldots, k_\ell^{(x_\ell)}$ provides no information except for $C(x)$." Here, we assume that the circuit $C$ is fixed and known (it represents the function $f_B$ that Bob is trying to compute), so intuitively what is being hidden here are the actual values of $x$, and any intermediate values computed by the circuit. To make this formal, we use a simulator-based definition, as follows.

**Definition 1.1.** *A garbled circuit is a pair of PPT algorithms* $(\mathsf{Garble}, \mathsf{Eval})$ *with the following two properties.*

- **(Correctness.)** *For any circuit* $C : \{0,1\}^\ell \to \{0,1\}$ *and any* $x \in \{0,1\}^\ell$,

$$\Pr_{(\widetilde{C}, k_1^{(0)}, k_1^{(1)}, \ldots, k_\ell^{(0)}, k_\ell^{(1)}) \leftarrow \mathsf{Garble}(1^n, C)}[\mathsf{Eval}(\widetilde{C}, k_1^{(x_1)}, \ldots, k_\ell^{(x_\ell)}) = C(x)] \geq 1 - \mathrm{negl}(n) \; .$$

  *(Here, we have chosen to allow some small probability of failure for convenience.)*

- **(Security.)** *There exists a PPT simulator* $S$ *such that for any circuit* $C : \{0,1\}^\ell \to \{0,1\}$ *and any* $x \in \{0,1\}^\ell$ *and any PPT adversary* $\mathcal{A}$,[1]

$$\Pr[\mathcal{A}(1^n, S(1^n, C, C(x))) = 1] - \Pr_{(\widetilde{C}, k_1^{(0)}, k_1^{(1)}, \ldots, k_\ell^{(0)}, k_\ell^{(1)}) \leftarrow \mathsf{Garble}(1^n, C)}[\mathcal{A}(1^n, \widetilde{C}, k_1^{(x_1)}, \ldots, k_\ell^{(x_\ell)}) = 1] \leq \varepsilon(n) \; .$$

Notice that in the security definition, we give the simulator access to the circuit $C$ as well as the output $C(x)$, so the garbling of the circuit is not required to hide the circuit $C$ itself. (Our construction actually will hide the *gates* of the circuit, though not the *topology*. I.e., the garbling will not hide the edges between gates, but it will hide the function that the gates compute. We will use this as part of the security proof, but we will not need this stronger property.)

It might not be clear why garbled circuits are useful for two-party computation (or why they're useful for anything at all!). In particular, if Alice sends Bob (1) a garbled circuit $\widetilde{C}$ corresponding to the function $f_B$ that Bob wishes to compute, (2) the garblings corresponding to her inputs $k_{1,A}^{(x_{1,A})}, \ldots, k_{\ell_A,A}^{(x_{\ell_A,A})}$, *and* (3) the garblings corresponding to Bob's inputs $k_{1,B}^{(x_{1,B})}, \ldots, k_{\ell_B,B}^{(x_{\ell_B,B})}$, then

[1]There's a subtlety here regarding uniform vs. non-uniform adversaries that I'm ignoring. In the uniform setting, we should really have $\mathcal{A}$ *choose* the input $x \in \{0,1\}^\ell$. The definition as written is the correct definition against non-uniform adversaries.

Bob can use the $\mathsf{Eval}$ function to compute $f_B(x_A, x_B)$. The security property of garbled circuits guarantees that "Bob learns nothing else." But, Alice cannot send Bob $k_{i,B}^{(x_{i,B})}$ in the clear without knowing $x_{i,b}$. She could send Bob both $k_{i,B}^{(0)}$ and $k_{i,B}^{(1)}$, but then we would lose security. So, to preserve both party's security, we will need some way for Alice to communicate $k_{i,B}^{(x_{i,B})}$ (and *not* $k_{i,B}^{(1-x_{i,b})}$!) to Bob without letting her learn $x_{i,B}$. We will do this using oblivious transfer.

## 2 Yao's protocol

We will now show how to use garbled circuits together with oblivious transfer to build a secure two-party computation protocol. We will then build garbled circuits. (We will see that garbled circuits are actually implied by one-way functions, so in some sense, the oblivious transfer is doing all of the heavy lifting in this protocol.) Just like in the previous lecture, we will assume for simplicity that the functionality has the form $f(x_A, x_B) = (\perp, f_B(x_A, x_B))$ where $f_B$ outputs a single bit. I.e., we assume that only Bob receives output and that this output consists of a single bit. (The single-bit restriction is not truly necessary. We could have instead simply defined garbled circuits with multi-bit output. The assumption that only Bob receives output can be removed by running the protocol twice, once to let Bob compute $f_B$ and once to let Alice compute $f_A$.)

We assume that we have access to a (one-out-of-two) oblivious transfer protocol that works for many-bit plaintexts $m_0, m_1 \in \{0,1\}^m$ for arbitrary (polynomially bounded) $m$, rather than just one-bit plaintexts like we considered earlier. I.e., Bob receives $m_b$ from Alice's pair of plaintexts $m_0, m_1 \in \{0,1\}^m$. In our current setting of honest-but-curious adversaries, this is without loss of generality, because we can always run a one-bit OT protocol many times in order to send many bits. (This does not work in the malicious setting because a malicious Bob could use such a protocol to learn, e.g., half the bits of $m_0$ and half the bits of $m_1$.)

With this, the protocol is relatively straightforward. Intuitively, Alice will garble a circuit $C$ representing the function $f_B(x_A, x_B)$, and she will then send Bob the garbled circuit $\widetilde{C}$ together with garblings of her own input $k_{1,A}^{(x_{1,A})}, \ldots, k_{\ell_A,A}^{(x_{\ell_A,A})}$. (She doesn't use oblivious transfer or anything else fancy to send these. She just sends them in the clear. Notice that while Bob learns $k_{i,A}^{(x_{i,A})}$, he *does not* learn $x_{i,A}$. In particular, he does not know whether this is $k_{i,A}^{(0)}$ or $k_{i,A}^{(1)}$.) Then, for each bit $x_{i,B}$ of Bob's input, they engage in an oblivious transfer protocol where Alice takes $m_0 := k_{i,B}^{(0)}, m_1 := k_{i,B}^{(1)}$ and Bob takes $b := x_{i,B}$. Then, Bob will learn $k_{i,B}^{(x_{i,B})}$. Finally, Bob uses the $k_{i,A}^{(x_{i,A})}$, the $k_{i,B}^{(x_{i,B})}$, and $\widetilde{C}$ to compute $C(x)$ (by running $\mathsf{Eval}(\widetilde{C}, k_{1,A}^{(x_{1,A})}, \ldots, k_{\ell_A,A}^{(x_{\ell_A,A})}, k_{1,B}^{(x_{1,B})}, \ldots, k_{\ell_B,B}^{(x_{\ell_B,B})})$).

Here is the protocol written formally.

| **Alice** | | **Bob** |
|---|---|---|
| `INPUT:` $x_{1,A}, \ldots, x_{\ell_A,A}$ | | `INPUT:` $x_{1,B}, \ldots, x_{\ell_B,B}$ |
| $(\widetilde{C}, (k_{i,A}^{(0)}, k_{i,A}^{(1)}), (k_{i,B}^{(0)}, k_{i,B}^{(1)})) \leftarrow \mathsf{Garble}(1^n, C)$<br>$k'_{1,A} := k_{1,A}^{(x_{1,A})}, \ldots, k'_{\ell_A,A} := k_{\ell_A,A}^{(x_{\ell_A,A})}$ | | |
| | $\xrightarrow{\widetilde{C}, k'_{1,A}, \ldots, k'_{\ell_A,A}}$ | |
| `FOR` $i = 1, \ldots, \ell_B$: | | |
| | $\xleftrightarrow{k'_{i,B} \leftarrow \mathsf{OT}((m_0 = k_{i,B}^{(0)}, m_1 = k_{i,B}^{(1)}), b = x_{i,B})}$ | |
| | | `OUTPUT` $\mathsf{EVAL}(\widetilde{C}, k'_{i,A}, k'_{i,B})$ |

As we've written it here, this protocol requires Alice and Bob to exchange $1 + \ell_B \cdot m_{\mathsf{OT}}$ messages, where $\ell_B$ is the number of bits in Bob's input and $m_{\mathsf{OT}}$ is the number of messages required for a single OT (except for the "blindfolded transfer construction," all of the constructions that we have seen have $m_{\mathsf{OT}} = 2$, which is the best one can hope for). This is still much less than the number of messages needed for the GMW protocol, which required roughly $m_{\mathsf{OT}} \cdot \ell_G$, where $\ell_G$ is the number of gates in the circuit. However, by being slightly more clever and running the many OTs in parallel, we can bring the number of messages in Yao's protocol down to just $m_{\mathsf{OT}}$. (If we want to compute a generic functionality in which both Alice and Bob receive output, then the number of messages needed is just $m_{\mathsf{OT}} + 1$. In particular, 3 messages is the best one can hope for.)

The intuition for why this protocol is secure is that (1) by the security of the OT, "Alice learns nothing from this protocol;" and (2) by the security of the OT together with the security of the garbled circuit, "Bob only learns $C(x_A, x_B)$." Below, we sketch a proof.

**Theorem 2.1.** *Yao's protocol is secure (assuming that the garbled circuit and the OT are secure).*

*Proof sketch.* We need to construct a PPT simulator $S_A$ for Alice and a PPT simulator $S_B$ for Bob and show that the views produced are indistinguishable from the views of Alice and Bob respectively in a true run of the protocol.

Alice's simulator $S_A$ is quite straightforward. Specifically, $S_A$ takes as input $1^n$ and $x_A$. It then simply does everything that Alice does (computing the garbled circuit $\widetilde{C}$ and the garbled inputs $k_{i,X}^{(b)}$), except that it replaces the views produced during the OT protocols with simulations produced by the OT simulator $S_{A,\mathsf{OT}}$. (Remember that, since the OT protocol is secure, there must be a simulator for Alice that produces views indistinguishable from Alice's actual view in a run of the OT protocol with Bob.) One can then use a hybrid argument, swapping simulated OT views with true OT views one at a time, to show that the full simulated view is indistinguishable from Alice's true view in a full run of the protocol.

Bob's simulator $S_B$ is a bit more complicated. It first uses the simulator $S_{\mathsf{Garble}}$ guaranteed by the security of the garbled circuit to produce a simulated garbled circuit $\widetilde{C}$ and simulated garbled input $k'_{i,X}$. It then runs the simulator $S_{B,\mathsf{OT}}$ repeatedly to simulate each of the OT protocols (using for Bob's output the $k'_{i,B}$ computed by $S_{\mathsf{Garble}}$). Finally, it outputs the evaluation. To prove that the resulting view is indistinguishable, we first switch to a hybrid in which the simulated garbled circuit $\widetilde{C}$ and simulated garbled input $k'_{i,X}$ are replaced by honestly computed values. A simple reduction from security of the garbled circuit shows that this hybrid is indistinguishable from the original. We then use another hybrid argument in which we replace the simulated OT views with true OT views one at a time. $\square$

## 3 Building garbled circuits

Yao's idea for building garbled circuits works as follows. For every gate $G$ in the circuit (including input gates), we sample two bit strings $k_G^{(0)}, k_G^{(1)}$. These are sometimes called keys or tags. (I am being deliberately vague about how we sample the $k_G^{(b)}$. They will end up being independently sampled secret keys for some secret-key encryption scheme.) We think of the tags $k_G^{(b)}$ as representing the gate $G$ taking the value $b$. In particular, the garbled input $k_1^{(0)}, k_1^{(1)}, \ldots, k_\ell^{(0)}, k_\ell^{(1)}$ are exactly the tags corresponding to the input gates. So, that tells us how to garble the input.

The garbled circuit $\widetilde{C}$ itself should somehow provide Bob with enough information so that, given $k_{G_1}^{(b_1)}$ and $k_{G_2}^{(b_2)}$ for parent gates $G_1, G_2$ of a gate $G$, Bob should be able to compute $k_G^{(G(b_1,b_2))}$. I.e., Bob should be able to compute the tags $k_G^{(b_G)}$ corresponding to the values $b_G$ taken by each gate $G$ in a computation of the circuit (but no other tags!). If we could do this, then by setting the final output tags to be $k_{\mathsf{out}}^{(0)} = 0$ and $k_{\mathsf{out}}^{(1)} = 1$, we will at least obtain a correct garbling scheme, and hopefully if we do it carefully, a secure scheme as well.

So, we just have to show how to garble a single gate. There are a few different but more-or-less equivalent ways to do this. Here's one. We will take the tags $k_G^{(b)} \leftarrow \mathsf{Gen}(1^n)$ to be independently sampled secret keys for some secret-key encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$. (We will need one small additional property of $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ later.) Then, the garbling of the gate $G$ contains *four* ciphertexts $c_G^{(0,0)}, c_G^{(0,1)}, c_G^{(1,0)}, c_G^{(1,1)}$ corresponding to the four different possible inputs to the gate $G$. We will set $c_G^{(b_1,b_2)} \leftarrow \mathsf{Enc}(k_{G_1}^{(b_1)}, \mathsf{Enc}(k_{G_2}^{(b_2)}, k_G^{(G(b_1,b_2))}))$. In other words, $c_G^{(b_1,b_2)}$ is an encryption of the key $k_G^{(G(b_1,b_2))}$ corresponding to the output $G(b_1, b_2)$ of the gate on input $(b_1, b_2)$, where the encryption uses the keys $k_{G_1}^{(b_1)}$ and $k_{G_2}^{(b_2)}$ corresponding to the input bits $b_1, b_2$.

Now, to evaluate the circuit, intuitively Bob just needs to use his keys $k_{G_1}^{(b_1)}$ and $k_{G_2}^{(b_2)}$ to decrypt $c_G^{(b_1,b_2)}$, giving him $k_G^{G(b_1,b_2)}$.

There are two related subtle issues, though: (1) Bob needs to know which ciphertext $c_G^{(b_1,b_2)}$ to decrypt; and (2) the order of the ciphertexts $c_G^{(b_1,b_2)}$ should not reveal anything about the values $b_1, b_2$ or $G(b_1, b_2)$. For example, suppose that Bob has two input keys $k_1, k_2$ (remember that Bob does not know and should not know whether, e.g., $k_1 = k_{G_1}^{(0)}$ or $k_1 = k_{G_1}^{(1)}$) and computes

$$\mathsf{Dec}(k_1, \mathsf{Dec}(k_2, c_G^{(0,0)})), \mathsf{Dec}(k_1, \mathsf{Dec}(k_2, c_G^{(0,1)})), \mathsf{Dec}(k_1, \mathsf{Dec}(k_2, c_G^{(1,0)})), \mathsf{Dec}(k_1, \mathsf{Dec}(k_2, c_G^{(1,1)}))\ .$$

(Notice that Bob uses the same pair of keys $k_1, k_2$ for each ciphertext, since he only has one pair of keys.) Depending on the specific encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ that we use, this could cause a number of problems. First of all, what if when Bob decrypts all of these ciphertexts, he receives as output four different bit strings $k'_{0,0}, \ldots, k'_{1,1}$, each of which looks like a valid secret key for our encryption scheme. How will he know which to choose?

To prevent this, we make sure to use a special encryption scheme $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ with the following extra property. For all $m$,

$$\Pr_{sk,sk' \leftarrow \mathsf{Gen}(1^n)}[\mathsf{Dec}(sk', \mathsf{Enc}(sk, m)) \neq \bot] \leq 2^{-n} \ .$$

In other words, with all but negligible probability, if we encrypt a message with one key and try to decrypt the resulting ciphertext with another, the decryption will fail. We can construct such a scheme easily by appending a uniformly random string $r \sim \{0,1\}^n$ to the secret key, and appending this same random string $r$ to every ciphertext. Then, on input $(sk, r)$, $(c, r')$, the decryption algorithm can simply check if $r = r'$. If not, it outputs $\bot$. Otherwise, it decrypts $c$ as normal. (In the context of the garbled circuit, this is like including along with each key an additional uniformly random string which is then used to match the key with its ciphertexts. There are other ways to do this that are more efficient and arguably more elegant, but we stick with this one because it makes the (already quite messy) notation a bit less messy.)

If we use such an encryption scheme, then (except with negligible probability), exactly one of Bob's decryptions will succeed, and he will know that the resulting output is the right tag for the gate $G$. However, if Alice literally put the ciphertexts in the order $c_G^{(0,0)}, c_G^{(0,1)}, c_G^{(1,0)}, c_G^{(1,1)}$, then this would lead to a new problem: Bob would learn the bits $b_1, b_2$ by seeing which of the ciphertexts $c_G^{(b_1,b_2)}$ he can decrypt.

So, we put the ciphertexts in a random order.

Here is the full scheme. We let $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be an encryption scheme with the special property described above. The $\mathsf{Garble}$ algorithm (run by Alice) samples two secret keys $k_G^{(0)}, k_G^{(1)} \leftarrow \mathsf{Gen}(1^n)$ for each gate $G$ in the circuit $C$, except for the output gate, where we simply take $k_{\mathsf{out}}^{(0)} = 0, k_{\mathsf{out}}^{(1)} = 1$. It takes the garblings of the input $k_i^{(b)}$ to be the corresponding keys for the appropriate input gates. Then, for each gate $G$ with parent gates $G_1, G_2$, it sets

$$\begin{aligned}
c_G^{(0,0)} &\leftarrow \mathsf{Enc}(k_{G_1}^{(0)}, \mathsf{Enc}(k_{G_2}^{(0)}, k_G^{(G(0,0))})) \\
c_G^{(0,1)} &\leftarrow \mathsf{Enc}(k_{G_1}^{(0)}, \mathsf{Enc}(k_{G_2}^{(1)}, k_G^{(G(0,1))})) \\
c_G^{(1,0)} &\leftarrow \mathsf{Enc}(k_{G_1}^{(1)}, \mathsf{Enc}(k_{G_2}^{(0)}, k_G^{(G(1,0))})) \\
c_G^{(1,1)} &\leftarrow \mathsf{Enc}(k_{G_1}^{(1)}, \mathsf{Enc}(k_{G_2}^{(1)}, k_G^{(G(1,1))})) \ .
\end{aligned}$$

Finally, it takes $T_G$ to be a four-tuple containing the four ciphertexts $c_G^{(b_1,b_2)}$ in a random order. It outputs $\widetilde{C} := (T_G)_{G \in C}$ (in other words, $\widetilde{C}$ consists of each garbled gate $T_G$ in some appropriate order).

The $\mathsf{Eval}$ function (run by Bob) behaves as follows. It takes as input $\widetilde{C} = (T_G)_{G \in C}$ and input tags $k_{\mathsf{in}_1}, \ldots, k_{\mathsf{in}_\ell}$, one for each input gate. Then, for each gate $G$ with parents $G_1$, $G_2$, it attempts to decrypt each of the four ciphertexts $c_G^{(b_1,b_2)}$ in $T_G$ using the keys $k_{G_1}, k_{G_2}$. With high probability, only one of these decryptions succeeds, yielding a new key $k_G$. Eventually, it does this for the output gate, yielding the output of the circuit, $k_{\mathsf{out}} = 0$ or $k_{\mathsf{out}} = 1$.

## 3.1 Security

Correctness of this garbling scheme is clear. (In particular, by applying union bound, we see that Bob's decryption will succeed for exactly one ciphertext for each gate except with negligible probability.)

Intuitively, the scheme is secure because (1) by the semantic security of the encryption scheme "Bob only learns one plaintext per gate"; and (2) since we randomized the order of the ciphertexts $c_G^{(b_1,b_2)}$, "he does not learn anything from the position of the ciphertext that he is able to decrypt."

To make this formal, we need to create a simulator $S$ that simulates the entire garbled circuit $\widetilde{C}$ together with garbled input $k_1 = k_1^{(x_1)}, \ldots, k_\ell = k_\ell^{x_\ell}$, knowing only the circuit $C$ and the output $y = C(x)$ (but not $x$!). Put succinctly, the simulator is quite simple: it replaces all of the gates in $C$ with gates of the form $G(b_1, b_2) = y$ for all $b_1, b_2 \in \{0, 1\}$. It then garbles this circuit, and takes the input keys to be $k_1' := k_1^{(0)}, \ldots, k_\ell' := k_\ell^{(0)}$, i.e., it assumes that the input is $0^\ell$. (There are many other ways to do this as well. E.g., one can replace all but one ciphertext in each gate by an encryption of zero and choose $k_1', \ldots, k_\ell'$ so that these "fake" ciphertexts are never decrypted. Or, one can notice that only the output gate $\mathsf{out}$ matters, and the simulator therefore does not need to set *all* gates to output $y$. More generally, one can choose to garble any circuit $C'$—with the same topology of gates—and input $x'$ such that $C'(x') = y$.)

Intuitively, the resulting output $\widetilde{C}'$ and $k_1', \ldots, k_\ell'$ are indistinguishable from honestly generated $\widetilde{C}$ and $k_1^{(x_1)}, \ldots, k_\ell^{(x_\ell)}$ because the only difference is what is encrypted in the unopened ciphertexts. (Make sure you see that this is true. The above succinct description is hiding some subtlety.) So, intuitively, semantic security of the encryption scheme should imply that the two garbled circuits are indistinguishable.

The issue that we run into is that there are dependencies between the various ciphertexts. In particular, parent gates in $\widetilde{C}$ contain encryptions of the keys used in the child gates. To solve this, we use a careful hybrid argument (of course) in which we switch the gates in $\widetilde{C}'$ to those in $\widetilde{C}$, one at a time. (Really, we will have *two* hybrids per gate, since we need to worry about the security of the *double* encryptions $c_G^{(b_1,b_2)}$, but we skip this formality.) We must be careful of two things: (1) that we maintain the output behavior of the circuit throughout; and (2) that we order the hybrids in such a way that we always switch a simulated child gate with a real child gate *before* we switch its parent gates.

Actually, it is convenient to first switch the circuit $\widetilde{C}'$ with the circuit $\widetilde{C}''$ in which instead of taking each gate to be the constant gate that always outputs $y$, we replace gate $G$ with the constant gate that always outputs $b_G(x_1, \ldots, x_\ell)$, where $b_G(x_1, \ldots, x_\ell)$ is the value that the gate $G$ takes when the circuit is run on input $x_1, \ldots, x_\ell$. Of course, the simulator does not know $b_G(x_1, \ldots, x_\ell)$, but if you think about it, you should be able to see that these two garbled circuits are identically distributed. In particular, our garbled circuit construction yields the exact same distribution for any two circuits whose gates are all constant, provided that the output gate is the same. We also replace $k_1' = k_1^{(0)}, \ldots, k_\ell' = k_\ell^{(\ell)}$ with $k_1'' = k_1^{(x_1)}, \ldots, k_\ell'' = k_\ell^{(x_\ell)}$. Again, the distributions are identical.

Now, let's look at the fist two hybrids, hybrid 1 and hybrid 2. In hybrid 1, the garbled circuit is just $\widetilde{C}''$. In hybrid 2, we change the output gate from the constant $y$ gate to the actual output gate $\mathsf{out}$ in the circuit $C$. In the reduction showing that hybrid 1 is indistinguishable from hybrid 2, we of course reduce from semantic security of the encryption algorithm. Specifically, we first switch encryptions under $k_{G_1}^{(1-b_1)}$ from those in hybrid 1 to those in hybrid 2, where $G_1$ is the first parent of $\mathsf{out}$. We then switch encryptions under $k_{G_2}^{(1-b_2)}$. Crucially, in each of these steps, the reduction can produce everything in the garbled circuit itself *except* for the ciphertexts that are encrypted under these keys. These ciphertexts are provided by the challenger in the semantic security game. In particular, the keys $k_{G_1}^{(1-b_1)}$ and $k_{G_2}^{(1-b_2)}$ are independent of the rest of the garbled circuit because the gates $G_1$ and $G_2$ are constant, so that, e.g., the ciphertexts $c_{G_1}^{(b'_1,b'_2)}$ are all encryptions of $k_{G_1}^{(b_1)}$ and *not* $k_{G_1}^{(1-b_1)}$.

If we did the hybrids in the wrong order, then at some point the reduction would need to produce a ciphertext $c_{G_i}^{(b'_1,b'_2)}$ that is an encryption of a key $k_{G_i}^{(1-b_i)}$ that it does not know. This is why we must be so careful about the order.

The full proof just uses the above idea, but requires some tedious bookkeeping. So, we will elide it for simplicity. You should be able to convince yourself that you could convert the above into a full proof if you needed to.

## 4 A brief note on efficiency

Garbled circuits might seem rather complicated and not particularly efficient. But, in practice, they can be made quite efficient, and they are therefore actually used in practical applications. Efficient implementations of garbled circuits use the same high-level ideas as the construction that we described above, but they use many optimizations to greatly reduce the size of $\widetilde{C}$ and the running time of the $\mathsf{Eval}$ function.

Even without optimizing, the above protocol is far more efficient than GMW in practice, essentially because it replaces many OT operations with secret-key encryption and decryption. In practice, secret-key encryption and decryption are far far more efficient than oblivious transfer protocols (for comparable levels of security). (More generally, we think of secret-key primitives as far more efficient than public-key primitives—and we think of OT as a public-key primitive.) And, of course, all of this encryption and decryption happens *locally*, while the GMW protocol requires many messages to be passed back and forth between Alice and Bob.

But, further optimization makes the protocol quite practical. Perhaps the most important optimization is the Free XOR optimization [KS08], which shows how to "get XOR gates for free." In particular, if a gate $G$ is an XOR gate (i.e., $G(b_1,b_2) = b_1 \oplus b_2$), then one can set $k_G^{(b_1 \oplus b_2)} := k_{G_1}^{(b_1)} \oplus k_{G_2}^{(b_2)}$. One can prove that this this does not harm security (assuming that the secret-key encryption scheme is reasonable—e.g., it suffices if the key is simply a uniformly random bit strings). But, it means that the bit length of $\widetilde{C}$ does not grow at all with the number of XOR gates in the circuit, and the $\mathsf{Eval}$ function can compute XOR gates by simply performing bit-wise XOR on the keys, which is far more efficient than running a decryption algorithm. (Remember that a similar optimization was possible with the GMW protocol as well.)

This particular optimization means that we also care about the *structure* of the circuit. Specifically, it is far more efficient to garble circuits with fewer AND gates, so that it is preferable to find a circuit $C$ representing our function $f$ that has fewer AND gates. Indeed, there is by now a basically a whole industry devoted to finding circuits computing various functions $f$ with relatively few AND gates.

AND gates are still costly, but they can themselves also be optimized, e.g., using the Half And construction [ZRE15]. The resulting scheme requires just two ciphertexts for each AND gate and no ciphertexts at all for each XOR gate.

Other simple optimizations are important as well. E.g., the double encryption construction $c := \mathsf{Enc}(k_1, \mathsf{Enc}(k_2, k_3))$ that we used above can be extremely wasteful, depending on the specific parameters of the scheme. It is often far more efficient to sample $r$ uniformly at random and set $c := (\mathsf{Enc}(k_1, r), \mathsf{Enc}(k_2, r \oplus k_3))$. And, our construction of special encryption was wasteful.

One can include random distinct two-bit labels with the ciphertexts to “let Bob know what to decrypt,” instead of the random $2n$-bit labels that we used. One also does not need to use a fully semantically secure secret-key encryption scheme, but rather only a secret-key encryption scheme that can “securely encrypt random messages,” which allows for further efficiency gains.

Indeed, there is a large body of literature studying optimizations of garbled circuits in various settings, and this is a very active area of research. E.g., it was recently discovered how to “use 1.5 ciphertexts per AND gate”, rather than two [RR21].

Optimizations are possible with oblivious transfer as well. For example, it is possible to perform *many* oblivious transfers *much* more efficiently than the naive method of running one oblivious transfer protocol many times [Bea96], using an idea called *OT extension*.

## References

[Bea96] Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *STOC*, 1996. 9

[BMR90] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *STOC*, April 1990. 2

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR Gates and applications. In *Automata, Languages and Programming*, 2008. 8

[RR21] Mike Rosulek and Lawrence Roy. Three halves make a whole? Beating the half-gates lower bound for garbled circuits. In *CRYPTO*, 2021. 9

[Yao86] A. C. Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167, October 1986. 2

[ZRE15] Samee Zahur, Mike Rosulek, and David Evans. Two Halves Make a Whole. In *EUROCRYPT*, 2015. 8