SECURE MULTI-PARTY COMPUTATION: GARBLED CIRCUITS

RYAN MORENO

1 Abstract

This paper presents Yao's *Garbled Circuit* protocol to compute any boolean function $F(a,b)$ where a and b are Alice and Bob's private inputs. Assuming Alice and Bob are honest, this protocol is guaranteed to produce the correct result without leaking any additional information about a or b. Yao's protocol is an example of Secure Multi-Party Computation, a process in which multiple parties with private inputs collectively compute a function without the use of a third-party. Before presenting the *Garbled Circuit* protocol, this paper introduces Secure Multi-Party Computation, explaining the canonical *Millionaire Problem*. We then examine *Oblivious Transfer*, a necessary tool in Yao's protocol.

2 Introduction

Secure Multi-Party Computation (SMPC) is the problem of computing a function $F(x_1,x_2...x_n)$ when the inputs are distributed among n actors, each having a piece of private data x_i. The function must be computed without using a third-party and without leaking any additional information about the actors' inputs. To formalize the notion of security, we will consider *real world* security and *ideal world* security. In the ideal world, we are allowed to use a trusted third-party to compute F. The SMPC protocol is considered the real world and is expected to match ideal security. This means that the SMPC protocol can leak any information about the actors' inputs that would have been leaked by simply learning the result of F. For example, consider the function $F(a,b) = a \wedge b$. If Alice has input $a = 1$, then Bob's input b is leaked by the result of $F(a,b)$; if the result is 1, we know Bob's input was 1, and if the result is 0, we know his input was 0. Even in the ideal world, Bob's input would be leaked by the result of the computation, so the SMPC protocol is allowed to leak this information as well. As a more abstract example, Alice could choose to leak her input by simply telling Bob what her input is. Because this could occur in the ideal world of a trusted third-party, the SMPC protocol is not expected to avoid this type of self-sabotaging leak.

Another factor to consider is the actors' level of honesty. The simplest level of security is *passive security*, which assumes that the actors will follow the protocol and not lie about intermediate results. Adversaries are expected to use leaked information to compute other actors' inputs, but we assume they won't actively cheat the protocol. This is a naïve assumption because in the real world corrupted parties might be willing to cheat the protocol. *Malicious security* assumes that the actors are willing to cheat the protocol, so SMPC protocols with malicious security must return the correct result or notify the actors if one of the individuals has cheated. In between these two levels of security is *covert security*, which ensures that if an actor diverges from the protocol, they will be caught with high probability. This is often more efficient than malicious security, and in practice can force adversaries to act passively due to outside economic or social incentive to not get caught cheating. In this paper, we only consider passive security, assuming the actors behave honestly.

This paper presents Yao's *Garbled Circuit* protocol to compute any boolean function on two parties' private inputs. In order to provide a gentle introduction to SMPC, we start by considering a solution to the *Millionaire Problem*, in which two millionaires want to know who has more money without releasing additional information about their wealth. Then we will examine *Oblivious Transfer*, a protocol allowing Bob to select one of Alice's many messages in such a way that Alice doesn't know which message Bob selected and Bob doesn't learn any of Alice's other messages. *Oblivious Transfer* is a necessary tool in the *Garbled Circuit* protocol. Finally, we will examine the *Garbled Circuit* protcol. In this protocol, Bob creates a computation table allowing Alice to calculate an arbitrary boolean function on their private inputs without learning any intermediate results.

3 The Millionaire Problem

3.1 Problem Description

Yao introduced the idea of SMPC by presenting the *Millionaire Problem*. Consider two millionaires who want to know who has more money without releasing additional information about their wealth. Effectively, we need to compute $A \geq B$. For example, let's say that Alice has wealth $A = \$7$ million and Bob has wealth $B = \$5$ million. Alice and Bob need to learn that Alice has more money. However, it's imperative that Alice learn nothing more about Bob's wealth than that he has less than \$7 million. Likewise, Bob must learn nothing more about Alice's wealth than that she has more than \$5 million. Yao describes a method for solving this problem, detailed below [1].

For simplicity, we will assume that $A, B \in [1,10]$. We will also assume that Alice has a public key (e, n) and a private key (d, n), which she can compute using the methods of RSA [2].

3.2 Protocol

To start the process, Bob chooses a random number x such that $|x| = |n|$ (this will be relevant later) and encrypts x using Alice's public key (e, n), resulting in a new value $c = \text{encrypt}(x)$. Bob then computes $m = (c - B + 1) \bmod n$ and sends m to Alice. Since the x that Bob originally chose was random, m looks random to Alice.

Alice now computes all possible options for x by trying every possible value of Bob's wealth B (which we restricted to $[1,10]$). Formally, Alice computes $x_i = \text{decrypt}(m+i-1)\ \forall i \in [1,10]$. Note that $x_B = \text{decrypt}(m+B-1) = \text{decrypt}(c-B+1+B-1) = \text{decrypt}(c) = x$, the original random value that Bob chose. However, to Alice, all x_i look random. Additionally, to Bob all x_i except x_B look random since Bob doesn't have access to Alice's decryption key and isn't able to explore the encrypted space.

Next, Alice chooses a random prime p such that $|p| = |n|/2$. She calculates $w_i = x_i \bmod p$ for each value x_i. Because p is half the length of x_i, taking $x_i \bmod p$ obfuscates the original values of x_i. This is important because Alice will be sending all values of w_i to Bob. If Alice sent back x_i directly, she would give Bob additional information about her private encryption key because the values of x_i represent Alice exploring the encryption space. Since Bob doesn't have access to Alice's decryption key, he shouldn't be able to walk around the encryption space and then decode the values.

Finally, Alice adds 1 to each value w_i where $i > A$. In this way, Alice has bumped each value of i that is greater than her own wealth. Alice sends p and all values of w_i to Bob. Since all values w_i look random to Bob except for w_B, Bob can't tell which values Alice bumped except for w_B, so he doesn't learn any information besides whether he has more money than Alice. To see if he has more money than Alice, Bob checks if $w_B = x \bmod p$ where x is his original random number. If the values are equal, then Alice didn't bump the value w_B, so we know Bob doesn't have more money than Alice, meaning $A \geq B$. If the values are not equal, then Alice bumped the value w_B, so we know Bob has more money than Alice, meaning $B > A$. Bob sends the result of the exchange back to Alice, so both know who has more money.

3.3 Correctness and Security

The protocol is correct because $A \geq B$ iff $w_B = x \bmod p$, so Bob is able to tell exactly who has more money. As for security, Alice only learns m which looks random to Alice since Bob computes it based on a random number x. So, Alice learns no information from this exchange. Bob learns $w_i\ \forall i \in [1,10]$. This information doesn't tell Bob anything extra about Alice's wealth. Since Bob can't walk around the encryption space, he can't calculate x_i when $i \neq B$. Therefore, he can't calculate the original values of w_i when $i \neq B$, so he can't tell which values of w_i Alice bumped. Additionally, Bob doesn't gain any information about Alice's private key from walking around the encryption space because he can't recover x_i from w_i due to the loss of information with $\bmod p$.

4 Oblivious Transfer

4.1 Problem Description

Oblivious Transfer (OT), developed by Even, Goldreich, and Lempel, is an important piece of many SMPC protocols [3]. The goal of OT is that Alice offers n different messages, and Bob selects and receives one of them. After the transfer process, Alice shouldn't know which message Bob chose, and Bob shouldn't gain any information about the other messages. Without loss of generality, we can consider each message to be a single bit. If Alice and Bob wanted to transfer longer messages, they could repeat the OT protocol for each bit.

OT makes the *Millionaire Problem* trivial because Alice can simply offer Bob 10 messages, one corresponding to each of possible value of Bob's wealth. Each message will contain a 0 or a 1, corresponding to "I'm wealthier ($A \geq B$)" or "You're wealthier ($A < B$). Shortly, we will see a more interesting application for OT.

4.2 Protocol

In the OT protocol, Alice first chooses a random trapdoor permutation consisting of a function, inverse function, and hard-core bit, $\{f, f^{-1}, B_f\}$. According to Diffie and Hellman, Alice can create a random trapdoor permutation such that given the function, it is infeasible to compute the inverse function [4]. Alice sends f and B_f to Bob, but keeps f^{-1} private. Bob chooses n random numbers x_1 ... x_n corresponding to the n messages Alice is offering. Bob computes $f(x_i)$ for whichever message i he would like to receive. He then sends $y_1 ... y_n$ to Alice, such that $\{y_1, y_2, ... y_i, ... y_n\} = \{x_1, x_2, ... f(x_i), ... x_n\}$. To Alice, these values look completely random. In particular, $\text{F}(x_i)$ still looks random because f is a permutation, so $\text{F}(\textit{random})$ yields a random value over the same domain.

Alice takes the hardcore bit of the inverse function of each value Bob sends such that $\{c_1 ... c_n\} = \{B_f(f^{-1}(y_1)) ... B_f(f^{-1}(y_n))\}$. Notably, $c_i = B_f(x_i)$, the hard-core bit of Bob's original random number. If Bob were to see the values $\{c_1 ... c_n\}$, they would all look random except for c_i. This is because Alice kept the inverse function of her random trapdoor permutation private, and it is impractical for Bob to compute the inverse function. Alice then computes $\{d_1 ... d_n\} = \{b_1 \oplus c_1 ... b_n \oplus c_n\}$ for each message b she is offering. Note that we originally restricted Alice's messages to be single-bit. We made this restriction so she can perform the xor of the hard-core bits she calculated with her messages. At this point, $d_i = b_i \oplus x_i$. To Bob, all d values except d_i look random because the xor function preserves the randomness of the c values. Alice sends the values $\{d_1 ... d_n\}$ to Bob, and Bob gets the ith message by computing $d_i \oplus x_i = b_i$.

4.3 Correctness and Security

As described in the protocol section, Bob ends up with the message b_i that he was expecting. Because Alice uses a random trapdoor permutation, all of the other d values Bob receives look random to him, so he can't calculate any of the other messages Alice offers. Alice doesn't gain any information about which message Bob chose because the only information she receives is $\{y_1 ... y_n\}$ which all look random to her. Even though Bob sent $y_i = f(x_i)$, f preserves randomness, so even with infinite computing power, Alice couldn't detect i.

It's worth noting that the security of this protocol depends on Bob being honest. If Bob wanted to decrypt all of Alice's messages instead of just one, he could cheat by computing $\{y_1 ... y_n\} = \{f(x_1) ... f(x_2)\}$.

5 Garbled Circuits

5.1 Problem Description

The culmination of this paper is a description of Yao's protocol to compute any boolean function $F(a,b)$ where Alice knows a, Bob knows b, and they both want to keep their inputs private. To present the basic idea of the protocol, consider the computation of $F(a,b)$, a single-bit, single-gate boolean function. Bob can simply compute $x_0 = F(0,b)$ and $x_1 = F(1,b)$. Using OT, Alice chooses to learn $x_a = F(a,b)$ and shares the answer with Bob. This may seem trivial, but consider the computation of $F(a,b) = a \wedge b$. If Bob has $b = 0$, then calculating $F(a,0) = a \wedge 0 = 0$ doesn't leak any information about a. On the other hand, if Bob has 1, then computing $F(a,1) = a \wedge 1 = a$ leaks a completely. However, this isn't a problem for this protocol, because it holds up to the security of the ideal world. Even if Alice and Bob passed their values to a trusted third-party to calculate $F(a,b)$, simply knowing the answer to $F(a,1) = a$ would leak a to Bob.

Unfortunately, this simple example doesn't scale to multiple bits or multiple gates because Alice and Bob would have to perform OT for each gate, leaking information about the output of each intermediate gate to Alice. To resolve this problem, we will keep all of the intermediate values encrypted.

5.2 Protocol

Initially, we will consider Alice and Bob computing a single-bit, single-gate function where the outputs are encoded. First, Bob creates the table shown in Figure 1. To fill in the table, Bob creates encryption schemes $S_1 = (E_1, D_1) ... S_6 = (E_6, D_6)$ and selects random values p, s, m, and u. In the table Bob creates, each of Alice and Bob's input values correspond to a specific encryption scheme (for example, in Figure 1, $a = 0$ corresponds to S_1). Note that S_3 and S_4 are assigned to Bob's possible input bits randomly, meaning S_3 is assigned to 0 with a probability of $1/2$, and S_4 is assigned the other value. Likewise, S_5 and S_6 are assigned to the possible output bits randomly. Alice will be given the decryption keys for the two encryption schemes corresponding to her input and Bob's input. Each row of the table represents an input pair (a,b). For example, row 2 of Figure 1 corresponds to $a = 0$ and $b = 1$. With access to D_1, corresponding to $a = 0$ and D_4, corresponding to $b = 1$, Alice would be able to decrypt the values s and t. We want row 2 to correspond to an output of 0 or 1 depending on what gate we're considering. For example, if the function is $F(a,b) = a \wedge b$, then we want row 2 to correspond to an output of $F(0,1) = 0 \wedge 1 = 0$, which has been assigned S_5. When Bob creates the table, he'll define t such that $s \oplus t = D_5$. This way, Alice will get the decryption key for S_5 iff she has s and t. Bob creates the table in this way, assigning q, t, n, and v so that each row corresponds to the appropriate output based on the boolean gate in question. See Table 1 for an example set of assignments.

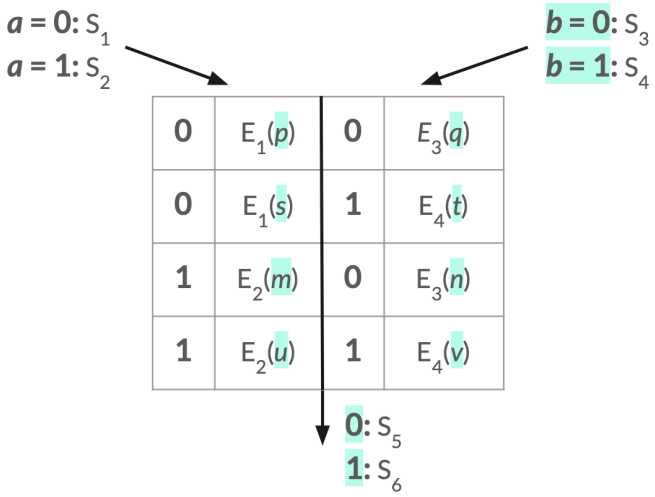

Figure 1: Computation table of a single-bit, single-gate boolean function (blue represents private to Bob)

Alice		Bob		output	
$a = 0$:	$E_1(p)$	$b = 0$:	$E_3(q)$	output 0:	$p \oplus q = D_5$
$a = 0$:	$E_1(s)$	$b = 1$:	$E_4(t)$	output 0:	$s \oplus t = D_5$
$a = 1$:	$E_2(m)$	$b = 0$:	$E_3(n)$	output 0:	$m \oplus n = D_5$
$a = 1$:	$E_2(u)$	$b = 1$:	$E_4(v)$	output 1:	$u \oplus v = D_6$

Table 1: Example assignments for $F(a,b) = a \wedge b$

Now that Bob has created the table, he permutes the rows randomly and sends it to Alice, keeping his inputs and the outputs private. Bob also sends D_3 or D_4 according to his input. Finally, Alice chooses either D_1 or D_2 (according to her input) using OT. Now Alice is able to use the two decryption keys to decrypt the pair of private values k, l in the row corresponding to her and Bob's input. Finally, she computes $D_i = k \oplus l$, which is either D_5 or D_6 depending on the gate in question. Note that at this point Alice has a new decryption key D_5 or D_6, but doesn't know what output (0 or 1) this corresponds to. Even though Alice can decrypt two other private values, these private values are from different rows, so she only gets half of the information necessary to compute the other decryption key, D_6 or D_5. This partial information is rendered useless because the output decryption keys are calculated using xor with another random, private value.

The point of keeping the output values private is that we can now link multiple gates together as in Figure 2 so that Alice can compute a large boolean function without learning intermediate results. Note that although Alice can't know which output value (0 or 1) she got from the first gate, she needs to know which decryption key she got (D_5 or D_6) so that she knows which row in the second table to decrypt. This can be accomplished by encoding the sentence "You got decryption key 5!!" using E_5 and the sentence "You got decryption key 6!!" using E_6. If Bob includes these encoded sentences with the table, then Alice can easily check which decryption key she got. She can then use the decryption key she received from the first table (corresponding to the first gate of the boolean function) as input to the second table (corresponding to the second gate of the boolean function) without Bob knowing which key she received. If she needs Bob's input to compute the second gate, she can get the new decryption key for the second table using OT. These tables can be strung together as many times as necessary in order to compute an arbitrarily large boolean function on inputs with many bits. When creating the table, Bob leaves the outputs of the final table unencoded. This way, Alice learns the output to the entire boolean function and can share the result with Bob.

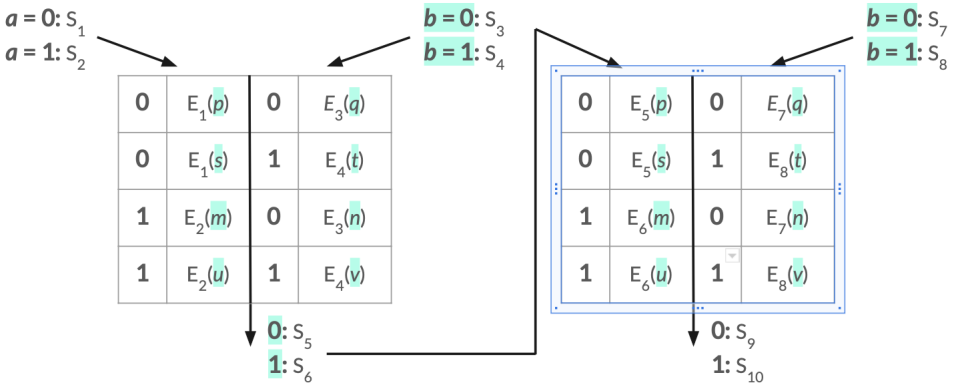

Figure 2: Computation table of a multi-bit, multi-gate boolean function
(blue represents private to Bob)

5.3 Correctness and Security

Any boolean function can be composed of $\wedge$ and $\vee$ gates on single bits. So, we can string together tables such that the final result of the protocol is $\mathrm{F}(a,b)$ for any boolean function F. During the protocol, Alice learns either D_3 or D_4, but this is uncorrelated with Bob's input. She learns only D_1 or D_2 according to her input. Using these values she can compute either D_5 or D_6, but cannot compute the other value because it is obfuscated by the xor function. Because the relation between the intermediate output bits and the output decryption keys are unknown to Alice, she doesn't learn the intermediate results of individual gates within the boolean function. Further, Alice only transfers the final answer to Bob, so he also doesn't learn any intermediate results. By following this protocol, Alice and Bob are able to jointly compute any boolean function $\mathrm{F}(a,b)$ without leaking any additional information about their inputs.

6 Discussion

6.1 Open Problems

An interesting area of research related to SMPC is *communication complexity*. In the context of SMPC, communication complexity asks what amount of communication overhead is necessary for secure computation. Current SMPC protocols for more than two parties can require exponential communication relative to the input length [7]. This exponential blow-up occurs because the protocols rely on representing the function they want to compute as a circuit, and some functions require exponentially large circuits. There is currently a large gap between the known lower-bounds of communication complexity (the minimum communication we know is required) and the known upper-bounds (the minimum amount of communication we can currently achieve) [7]. It will take further complexity research to reduce this gap and converge on the communication overhead strictly necessary for secure computation.

6.2 Applications

SMPC has practical applications outside of theoretical research. For example, consider voting. In the United States, we each cast our individual votes by giving them to a trusted third-party – the government – who computes which candidate wins. However, the majority of Americans believe there is significant election fraud, suggesting that we may not have a trusted third party [5]. Although the protocol presented in this paper doesn't scale well for computing an election, there are practical SMPC protocols to correctly compute the outcome of an election without relying on a trusted third-party [6].

Private bidding and auctions is another situation that lends itself to SMPC. Individual participants provide input in the form of how much they are willing to bid for an item. The goal is to compute who won the item without releasing information about how much any participant offered to pay. This is similar to the *Millionaire Problem*, except that there are more parties involved. Using SMPC, private auctions can be accomplished without releasing sensitive data to a third-party. There are many other practical applications; anything that involves using a third-party to compute a result given sensitive participant data is a good candidate for SMPC.

7 References

[1] A. C. Yao, "Protocols for secure computations," *SFCS '82 Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, pp. 160–164, 1982.

[2] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Commun. ACM*, vol. 21, no. 2, pp. 120–126, Feb. 1978.

[3] S. Even, O. Goldreich, and A. Lempel, "A randomized protocol for signing contracts," *Commun. ACM*, vol. 28, no. 6, pp. 637–647, Jun. 1985.

[4] W. Diffie and M. Hellman, "New directions in cryptography," *IEEE Trans. Inf. Theor.*, vol. 22, no. 6, pp. 644–654, Sep. 2006.

[5] M. Rourke, "Views on the american election process and perceptions of voter fraud," *The Associated Press-NORC Center for Public Affairs Research*, 2016.

[6] J. Bermúdez, "A practical multi-party computation algorithm for a secure distributed online voting system," 2018.

[7] V. Vaikuntanathan, "Some open problems in information-theoretic cryptography," *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2017)*, vol. 93, 5:1–5:7, 2018.