Semi-honest Multi-Party Computation for n Parties

Kyle Reyes
Brown University
Providence, USA
kyle_reyes@brown.edu

Neil Ramaswamy
Brown University
Providence, USA
neil_ramaswamy@brown.edu

1 OVERVIEW

Secure multi-party computation (MPC) is a subfield of cryptography that focuses on securely computing the output of a function that requires inputs from multiple parties, without revealing any party's inputs to any other party. For our final project for CSCI 1515 (taught by Prof. Peihan Miao), we implement the Goldreich-Micali-Wigderson (GMW) protocol for semi-honest multi-party computation among any number of parties to compute a boolean function.

The GMW protocol allows for secure MPC on any arbitrary boolean circuit. For each input wire, one out of n semi-honest parties is responsible for its value. Here, semi-honest means that each party follows the protocol accordingly. The GMW protocol also requires that each of the n parties has an additive secret share for each input wire. That is, the sum of all the secret shares for a given wire should be the desired input for that wire.

This report focuses more on the implementation of this protocol, which can be found in this repository. More information about GMW can be found in Section 3.2 of *A Pragmatic Introduction to Secure Multi-Party Computation* [2]. To gain a full understanding of this project, one should also familiarize themselves with the Yao's project from this course, as this was largely built off of that assignment.

2 IMPLEMENTATION DETAILS

In all of the following sections, we use the following conventions:

(1) There are n parties, denoted P_1 through P_n
(2) A wire u has value s_u, which is shared to all parties. A share for a particular party i is denoted as s_{u_i}.

2.1 Communication and Channel Security

It has been noted that secure channels between peers is not strictly necessary in GMW if only one party needs to learn the output at the end. This is because GMW with n parties guarantees that the final output is only learned if all n secret shares for a wire are shared; thus, if only $n-1$ values are sent to one party who reveals the final answer, an eavesdropper can't learn anything about the (cumulative) secret share.

However, in our implementation, we will perform the Diffie-Hellman key exchange for each pair of parties. This will allow us to have authenticated encryption between any two parties (using the `CryptoDriver::encrypt_and_tag` method). As a result, every party can learn the output at the end.

2.2 Establishing Communication Deadlock-Free

When doing peer-to-peer communication with n parties, care must given to avoid deadlocks. If some parties i and j (zero-indexed) are trying to send messages to each other and their TCP buffers are both full, then both parties will hang; since no read calls are being made, a deadlock will ensue.

To prevent this issue, we devised a rule to allow parties to synchronously communicate without deadlocks. Consider some party i trying to communicate with party j. If $i < j$, then i will always send a message to j first, who should be ready to read; after j receives the message, it can respond to i.

Thus, if we want to do n-to-n communication for key exchange, for example, party i does the following:

(1) It makes i calls to `NetworkDriver::socket_read`, starting from party 0 to $i-1$ (inclusive).
(2) Then, it makes $(n-i)-1$ calls to `NetworkDriver::socket_send`, starting from party $i+1$ to $n-1$ (inclusive).

The same logic can be applied to establishing all the different TCP connections: simply swap `NetworkDriver::socket_read` with `NetworkDriver::accept`.

Every party also needs a way to know which socket corresponds to which party index. While this could be accomplished by having each party broadcast a message with their party index once connections have been made, the party index to socket mapping can be easily determined by any party. We know that the k-th successful call to `NetworkDriver::socket_read` must be from party k; we also know that the m-th successful call to `NetworkDriver::socket_send` must be from party $(x+1)+m$, where x is the current party's index. Each instance of pointwise communication is represented by a `PeerLink`, which stores the AES key, the HMAC key, and a private "socket" variable. This class also implements communication methods.

This pattern of communication, where a party at a higher index waits for a message from a lower index, is used both for key exchange and the initial secret sharing. It is pictured in Figure 1.

2.3 Agreeing on the Initial Secret Shares

Suppose we have m initial wires with indices 0 through $m-1$. Each party will initialize a vector called `shares` that holds the shares for each wire (in the entire circuit). Thus, if there are p wires in this circuit ($p > m$), `shares` will be of length p. In this section, we describe how to populate the shares for the initial set of m wires.

We use a shared configuration file that says which parties control which initial wires; of course, this means that a party can read the inputs of another party, since because all parties are running locally on the same filesystem. However, we do not worry about this case, because our local environment is necessarily an improper simulation of MPC. Thus, for each party P_i, starting from wire 0 to wire $m-1$, we do the following:

(1) If P_i owns the input for wire w, they need to create secret shares for s_w. They do so by generating $n-1$ random bits

for the other parties and setting their own share to be $s_{w_i} = s_w \oplus (\oplus_{k \neq i} s_{w_k})$. P_i then sends the shares to parties P_1 through P_n (in that particular order), excluding itself.

(2) Otherwise, P_i does not own the input for wire w. Suppose P_o is the owner of input w. In this case, P_i will start a `NetworkDriver::read` call from P_o.

2.4 Computing One Gate

Once all the parties have exchanged their initial shares, they can start to evaluate each gate. Note that the circuit must be topologically sorted (which happens in Yao's). We focus on evaluating one particular gate g by some party P_i in this section. Note that because any boolean circuit can be represented using XOR, AND, and NOT gates, we only need to show how the protocol works for these gates.

2.4.1 XOR Gates. For a particular gate g, it will have a g_ℓ, g_r, and g_o, which are the left-hand side, right-hand side, and output indices into the `shares` vector, described in Section 2.3. We can simply XOR `shares`$[g_\ell]$ with `shares`$[g_r]$, and put that value into `shares`$[g_o]$.

2.4.2 AND Gates. Readers should familiarize themselves with Section 2.1 of *Secure Multi-Party Computation of Boolean Circuits with Applications to Privacy in On-Line Marketplaces* before reading further [1].

To compute an AND gate, every party must communicate with every other party. Consider the following protocol with $n-1$ rounds, 1-indexed. At round k, party P_k will act as a sender in the 1-out-of-4 OT with every party $j > k$ (and, naturally, P_j will be the OT receiver). P_k does not need to OT with any party less than k, because the OT would have already been completed with that party previously.

After a party P_i completes this reshare step, they can compute their output wire s_{g_o}. Specifically, we need $a + b = s_{u_i} s_{v_j} + s_{v_i} s_{u_j}$, where a is for Party i and b for Party j. Suppose $i < j$, and Party i is the sender. Party i generates random a and generates

- $-a$ ($s_{v_j} = s_{u_j} = 0$). This is choice 0.
- $-a + s_{v_i}$ ($s_{v_j} = 0$, $s_{u_j} = 1$). This is choice 1.
- $-a + s_{u_i}$ ($s_{v_j} = 1$, $s_{u_j} = 0$). This is choice 2.
- $-a + s_{u_i} + s_{v_i}$ ($s_{v_j} = s_{u_j} = 1$). This is choice 3.

2.4.3 NOT gates. NOT gates are not discussed by the book or paper, but can be implemented fairly easily. We observe that to invert a secret share that equal some boolean b, we just need to invert one secret share. Since we are in the semi-honest setting, we just make P_0 invert their share, and allow every other party P_i set their output wire to be their input wire.

3 RESULTS AND TESTING

As for testing, a lot of infrastructure from Yao's prove to be useful. First, perform some manual unit testing on our `OTDriver` (which is now made to support 1-out-of-n OT) by slightly modifying "ot_test.cxx" from Yao's.

To verify the final output of our GMW implementation, we run the given circuits on the given inputs on our tested Yao's implementation, and save the output. We then partition the input wires across the n parties in the input file, ensuring that each input wire receives the same value as in Yao's. Then, we compare the output and ensure that they are equal. In addition to the individual gate tests, we ran

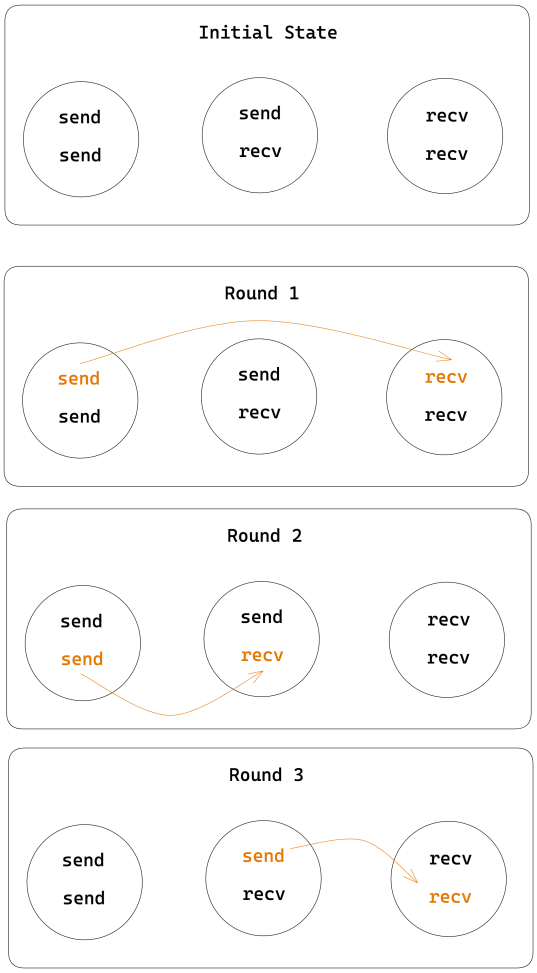

Figure 1: The deadlock-free sequential protocol for doing n-to-n party communication. Party i does i sends, and $(3-i)-1$ receive calls.

the "adder.txt" circuit with 3 and 5 parties and computed the same result as we did in Yao's.

4 FUTURE WORK

While this implementation does work, it is not the most efficient. Our client program is single-threaded, so communication is not parallelized. This explains why we need to be cautious with the order in which we send and receive messages between parties. Although this protocol is not trivial, the code pertaining to the protocol was relatively straightforward. In fact, attempting to produce a safe and scalable multithreaded solution proved to be the most difficult part. Due to time constraints, we abandoned this approach in favor of a solution that was much easier to debug. Should we have more time, migrating to a concurrent solution would be the next thing to do. If n is not too large, we can have a thread for each TCP connection. This does not scale too well, so ideally, one would have a thread pool. In this approach, we do not need to worry about the

order of sending and receiving messages. Once we have a more efficient implementation, we could try running “mult.txt" and “aes.txt", which would take 30 minutes to an hour to run with our current implementation.

Another improvement that can be made to the infrastructure is our input method. The expected format of the input file is that line i contains something of the form of “m:x”, which means that input wire i is controlled by party m with input value x. For the purposes of showing that the protocol works, this is fine; however, this defeats the purpose of secure MPC because each party can just inspect everyone else’s inputs. This can be fixed by having a configuration file, which specifies which party is responsible for each input wire, and a party-specific input file.

Other further work includes extending this protocol to a malicious setting, where parties may deviate from the protocol to find out as much information as they can. There are two ways to go about this: the GMW compiler and the Cut-and-Choose method.

REFERENCES

[1] Seung Geol Choi, Kyung-Wook Hwang, Jonathan Katz, Tal Malkin, and Dan Rubenstein. 2011. Secure Multi-Party Computation of Boolean Circuits with Applications to Privacy in On-Line Marketplaces. *IACR Cryptology ePrint Archive* 2011 (01 2011), 257.

[2] David Evans, Vladimir Kolesnikov, and Mike Rosulek. 2022. *A Pragmatic Introduction to Secure Multi-Party Computation.* NOW Publishers, Section 3.2, 37–42. https://securecomputation.org/docs/pragmaticmpc.pdf