A Note on Perfectly-Secure MPC with Amortized Linear Communication Cost

Zihan Hu Zhouzi Li Wenhao Wang

June 15, 2023

Abstract

In this project, we focus on perfectly-secure communication-efficient multi-party computation with guaranteed output delivery over point-to-point channels for corruption threshold $t < n/3$. We briefly review the protocol in [BTH08], which achieves the aforementioned security guarantee with communication complexity $O(Cn\kappa+D_M n^2\kappa+n^3\kappa)$ where C is the size of the circuit, n is the number of parties, D_M is the multiplicative depth of the circuit, and κ is the size of a field element. We design a new protocol with the following advantages:

- **Efficient:** It achieves the same security guarantee as [BTH08], but within communication complexity $O(Cn\kappa + n^3\kappa)$, which is as efficient as the protocol in [GLS19], the best-known protocol in terms of communication efficiency;
- **Simple:** Compared to the protocol in [GLS19], our protocol is simple to understand and describe.

1 Introduction

Secure multi-party computation (MPC) [Yao82, CCD88, GMW87] allows a group of n participants to evaluate a pre-agreed function securely, even if t of the participants are corrupted by an adversary. A semi-honest adversary can observe the actions of the corrupted participants and attempt to acquire unauthorized information. A malicious adversary, on the other hand, can go further and manipulate the corrupted participants to alter the computation outcome.

The MPC protocols differ in scenarios where the number of corrupted parties, the adversary type and the security level of the protocol are different. There are three different kinds of security, which are separately perfect security, unconditional security, and computational security. The perfect security requires a protocol to guarantee the success of the computation with no error probability against any information-theoretic adversary. If one allows small amount of error probability (e.g., negligible to the security parameter), we say this protocol achieves unconditional security. Computational security refers to protocols that rely on cryptographic assumptions and consider computationally bounded adversaries.

In the honest majority setting, $n \geq 2t+1$, where in the 1/3 corruption setting, $n \geq 3t+1$. Also there are different settings about the adversary type, and the MPC protocol can be against a semi-honest adversary or against a malicious adversary. Previous works have shown that in the honest majority setting, it is possible to construct a protocol that is unconditionally secure to compute any function against semi-honest adversaries, and that in the 1/3 corruption setting the protocol is perfectly secure against malicious adversaries [GBOW88]. It is known that 1/3 corruption setting is also necessary for the perfect security. Moreover, it is shown that if we think it tolerable for errors to happen with low probability, then the honest majority setting suffices to be secure against malicious adversaries. The honest majority security is known to be sufficient for unconditional security.

The origins of the MPC problem can be traced back to Yao's work in 1982 [Yao82]. Initial generic solutions, relying on cryptographic complexity assumptions and later on information-theoretic security, were largely inefficient, making them primarily of theoretical significance [CCD88, GMW87, GBOW88].

Afterwards more practical MPC protocols with perfect security for arbitrary computational functions emerged [BTH08, GLS19]. These protocols make MPC protocols efficient and possible to be implemented in real-life. There are many criteria of measuring the efficiency of an MPC protocol, such as communication complexity w.r.t. circuit size, computation complexity w.r.t. circuit size, etc.. In our work we will mainly focus on the communication complexity, which is measured in number of bits sent among different parties within the protocol specification. The communication complexity of the work [BTH08] is $O(Cn + D_M n^2 + n^3)$ fields elements, where C is the size of the circuit, n is the number of parties and D_M is the multiplicative depth of the circuit. The work of [GLS19] further improves the communication complexity to $O(Cn + n^3)$ field elements, which eliminates the overhead that includes the multiplicative depth. Yet the techniques included in [GLS19] to get rid of the overhead are complex and can be made simpler with proper modifications.

In our work, we introduce our perfectly secure MPC protocol that achieves $O(Cn + n^3)$ communication complexity with simpler techniques than [GLS19]. The settings of our protocol are exactly the same as those of [GLS19]. The high-level description of our protocol is given in section 2.

2 Technical Overview

In this section, we will start with a brief overview of the protocol in [BTH08], identify its key bottleneck, and show how to overcome the barrier to achieve better communication complexity. In the following, we will use n to denote the total number of parties and t to denote the corrupted threshold. We will always use C to denote the arithmetic circuit we are computing, D_M to denote the multiplicative depth of C, and κ to denote the size of a field element. We use $[x]_d$ to denote a degree-d Shamir secret sharing with secret x. It allows any $d+1$ parties to

reconstruct the secret x, but the shares of any d parties do not leak any information about x.

2.1 Review: The Protocol in [BTH08]

The main contribution of [BTH08] is a method to non-robustly but detectably generate $\Omega(n)$ uniform random sharings with perfect security and $O(n^2)$ communication. Combined with some techniques from [Bea92, HMP00, DN07], they can achieve communication cost $O(Cn\kappa + D_M n^2\kappa + n^3\kappa)$.

The protocol includes two phases called the preparation phase and the computation phase. In the preparation phase, we generate random double sharings to construct triples in batches, which will help us to compute the multiplication gate. In the computation phase, we consume the triples and evaluate the circuit gate by gate. Our main focus is the computation phase, so we will omit the details of the preparation phase and only emphasize that we can robustly prepare l triples in communication complexity $O(ln\kappa + n^3\kappa)$. Readers can refer to Appendix A for more details about the preparation phase.

Protocol 1. [BTH08]

1: Invoke the preparation protocol to get C Beaver triples $([a_k]_t, [b_k]_t, [c_k]_t)_{k=1}^{C}$ where a_k, b_k are independent uniform random values and $c_k = a_k b_k$

2: Evaluate the circuit gate by gate as follows

- Input Gate:
  1. For each input gate, we associate it with a triple $([a_k]_t, [b_k]_t, [c_k]_t)$
  2. All parties send their shares of $[a_k]_t$ to the dealer who is the owner of the input
  3. The dealer reconstructs a_k, and computes $d = x - a_k$ where x is the input
  4. Invoke Broadcast to let every dealer robustly broadcast the differences d in batches efficiently
  5. Each party computes $[x]_t = d + [a_k]_t$ as its input shares
- Random Gate: Use the sharings $[a_k]_t$ in a fresh triple $([a_k]_t, [b_k]_t, [c_k]_t)$
- Addition Gate: Each party locally computes the addition of their shares
- Multiplication Gate: Up to $\lfloor T/2 \rfloor$ ($T := n-2t$) multiplication gates are processed in batches. For multiplication gates in the batch, we denote the associated triples as $([a_k]_t, [b_k]_t, [c_k]_t)_{k=1}^{\lfloor T/2 \rfloor}$, the input sharings of the gates as $([x_k]_t, [y_k]_t)_{k=1}^{\lfloor T/2 \rfloor}$, and we compute the output sharings of the gates $([z_k]_t)_{k=1}^{\lfloor T/2 \rfloor}$ as follows

1. Each party locally computes $[d_k]_t = [x_k]_t - [a_k]_t, [e_k]_t = [y_k]_t - [b_k]_t$
2. Invoke ReconsPubl to publicly reconstruct the secrets $(d_k, e_k)_{k=1}^{\lfloor T/2 \rfloor}$
3. Each player locally computes $[z_k]_t = d_k e_k + d_k[b_k]_t + e_k[a_k]_t + [c_k]_t$

- Output gate: Each party sends his shares to the owner of the output, who reconstruct the output.

Here Broadcast can let each of n dealers robustly broadcast at most T elements in total communication complexity $O(n^3\kappa)$. It ensures that each honest party broadcasts the value he wants and that each honest party gets the same value no matter how corrupted parties deviate from the protocol. We only need each dealer to broadcast one element in step 4 for the input gate, so this step can be done in $O(n^3)$.

ReconsPubl is a protocol that can robustly reconstruct the secrets of up to T t-degree Shamir secret sharings in batches in total communication complexity $O(n^2\kappa)$. Roughly speaking, it uses Error Correction Code to encode the T secrets. So honest parties can always recover the secret even though corrupted parties may send incorrect values. For more details about the protocol, readers can refer to Appendix B.

Communication Complexity of Protocol 1 Apart from the $O(n^3\kappa)$ overhead in generating the triples and in distributing the inputs, Protocol 1 achieves linear amortized communication complexity per gate if each time there are enough multiplication gates to compute in batches. However, if a multiplication layer contains only a few gates (say only constant gates), we still need $O(n^2\kappa)$ to reconstruct the secrets before moving on to the next layer, causing $O(n^2\kappa)$ overhead per each multiplication layer. Thus the total communication complexity of Protocol 1 is $O(Cn\kappa + D_M n^2\kappa + n^3\kappa)$.

2.2 Our Protocol: Faulty Computation Before Batched Verification

2.2.1 A Naive Attempt and its Attack

As the $O(D_M n^2\kappa)$ overhead is incurred by the public reconstruction for each multiplication layer, our first attempt is to first let one party reconstruct the secret (which is $O(n\kappa)$ even when we don't have a batched of reconstructions), and then verify the reconstruction for gates from potentially different multiplication layers in batches using ReconsPubl. Formally, to compute $[xy]_t$, parties will go through the following steps:

1. All parties use a new Beaver triple $([a]_t, [b]_t, [c]_t)$
2. All parties locally compute $[x+a]_t = [x]_t + [a]_t, [y+b]_t = [y]_t + [b]_t$, and send the shares to P_{king}

3. P_{king} reconstructs $x + a, y + b$ and broadcasts the values

4. All parties compute $[xy]_t = (x + a)(y + b) - (x + a)[b]_t - (y + b)[a]_t + [c]_t$

And they will invoke ReconsPubl to verify the reconstruction only after a batch of multiplication gates.

However, this naive attempt may leak information to corrupted parties. To understand why this is the case, let's consider the following explicit attack.

Suppose we have sharings $[x]_t, [y]_t, [z]_t$ and we want to compute $(x \cdot y) \cdot z$.

If P_{king} is corrupted, he can deviate from the protocol by sending $x + a$ to $t + 1$ parties and sending $x + a + 1$ to the other $2t$ parties. Then after step 4, $t + 1$ parties hold shares of $[xy]_t$ while the other $2t$ parties hold shares of $[(x+1)y]_t$. Then when they compute the second multiplication gate, $t + 1$ parties will send shares of $[xy]_t + [a']_t$ to P_{king} while the other $2t$ parties will send shares of $[(x + 1)y]_t + [a']_t$ where $([a']_t, [b']_t, [c']_t)$ is the Beaver triple for the second multiplication gate. Thus P_{king} can obtain both $xy + a'$ and $(x+1)y + a'$, so he will get the intermediate result y, which should not be allowed.

2.2.2 Protecting the Sharings with a Mask

The above attack works because a corrupted party could potentially distribute arbitrary degree polynomials to other parties when he is supposed to broadcast, which will make the degree of the input wire for the next layer greater than t. Thus a degree-t Shamir secret sharing $[a]_t$ is not enough for protecting the secret.

In realizing this, our solution is to use degree-$(n-1)$ Shamir secret sharing $[a]_{n-1}$ to protect our sharings. Notice that double sharings $[a]_t, [a]_{n-1}$ can also be generated by first generating $[a]_t$, then generating degree-$(n - 1)$ random zero sharing $[0]_{n-1}$, and finally computing $[a]_t = [0]_{n-1} + [a]_t$. So we give the following procedure to compute $[xy]_t$ given $[x]_t, [y]_t$:

1. All parties use a new Beaver triple $([a]_t, [b]_t, [c]_t)$

2. All parties use two new degree-$(n - 1)$ random zero sharing $[0^{(1)}]_{n-1}, [0^{(2)}]_{n-1}$ (Random zero sharings can be generated in batches efficiently in a similar way as the generation of random sharings Appendix A.2. Specifically, each party P_i distributes a degree-$(n-1)$ random zero sharing $[0_{(i)}]_{n-1}$ to all other party. Then all parties locally apply a hyber-invertible matrix M on their shares and sends $2t$ of the resulting shares to $2t$ different parties to check whether it's a zero sharing. If the check passes, the remaining $n - 2t$ resulting shares satisfy our requirements.)

3. All parties locally compute two sharings $[x+a]_{n-1} = [x]_t + [a]_t + [0^{(1)}]_{n-1}$ and $[y + b]_{n-1} = [y]_t + [b]_t + [0^{(2)}]_{n-1}$, and send the shares to P_{king}

4. P_{king} reconstructs $x + a, y + b$ and broadcasts the values

5. All parties compute $[xy]_t = (x+a)(y+b) - (x+a)[b]_t - (y+b)[a]_t + [c]_t$

In this way, honest parties only send their shares after adding an independent random value as a mask. Thus they can evaluate multiple multiplication layers without leaking any information. However, the computation is faulty and we need to verify the computation carefully.

In the following section, we will follow the Player Elimination framework. Namely, it combines a procedure that does not leak any information but may be incorrect with two procedures called FaultDetection and FaultLocalization to verify whether the computation of the segment is correct and find two disputed parties (at least one of them is corrupted) if incorrect. In this way, we can kick out at least one corrupted party while maintaining the number of remaining corrupted parties t' does not exceed the corruption threshold each time the computation of the segment fails. So by dividing the whole computation into t segments and retrying a segment when the computation of the segment fails, we can achieve perfect security efficiently. More details about the Player Elimination framework are included in Section 3.1 for completeness.

We already present a way to evaluate the circuit gate by gate without leaking any information. In the remainder of this section, we will focus on how to verify the computation and how to detect the cheaters.

2.2.3 Verifying the Faulty Computation

After evaluating a batch of multiplication gates, it's time to run a verification procedure to tell us whether the computation is correct. We use the same method as [GLS19] to do the verification.

First, every party checks whether P_{king} sends the same value to them. This can be done based on a procedure called CheckConsistency, which can check whether a party P_{king} **broadcasts** T elements to all other parties, and if the check fails, find two disputed parties in communication complexity $O(n^2\kappa)$. As we have $\Omega(n)$ multiplication gates to be verified in batches, the amortized cost for each gate is $O(n)$.

Now we know that P_{king} sends consistent values. We only need to check that P_{king} sends correct values. This can be done by publicly reconstructing the values. As we have $\Omega(n)$ reconstructions to verify, ReconsPubl can efficiently reconstruct the correct values of $x+a, y+b$ from $[x+a]_t, [y+b]_t$ with amortized communication complexity $O(n\kappa)$ per reconstruction.

However, if the check fails, we need to identify whether (1) P_{king} is corrupted and he sends wrong values; or (2) P_{king} is honest and we should find another party who sends wrong shares of $[x+a]_{n-1}, [y+b]_{n-1}$, which results in the incorrect values. As degree-$(n-1)$ Shamir secret sharing does not have any redundancy, a change of only a single share can transfer a valid degree-$(n-1)$ Shamir secret sharing to

another valid one, which makes it difficult, even for honest P_{king}, to tell who causes the error.

2.2.4 Detecting the Cheaters

We will solve the problem by providing P_{king} with more necessary information in addition to the true whole sharing $[x+a]_t$ so that he can localize the error. In the following, we will focus on how to enable an honest P_{king} to find a corrupted party or two disputed parties who hold different opinions about the same message (and thus at least one of them is corrupted).

Without loss of generality, suppose the first wrong value reconstructed by P_{king} is $x^{(i)}+a^{(i)}$ (The i^{th} $x+a$ we need to reconstruct in this segment). Let $\overline{[x^{(i)}+a^{(i)}]}_{n-1}$ be the sharing P_{king} receives in the previous stage. If P_{king} is honest, he can localize the error as follows

1. All parties send their shares of $[x^{(i)}+a^{(i)}]_t$ to P_{king}
2. P_{king} uses the shares to recover the correct shares $[x^{(i)}+a^{(i)}]_t$
3. P_{king} computes $\overline{[x^{(i)}+a^{(i)}]}_{n-1} - [x^{(i)}+a^{(i)}]_t$, which is supposed to be a zero sharing $[0^{(i)}]_{n-1}$
4. All the parties send the randomness they use in constructing $[0^{(i)}]_{n-1}$ to P_{king}. Formally, for $j=1,2,\cdots,n$, P_j send his shares of $[0_{(k)}]_{n-1}(1\le k\le n)$ and the whole shares of $[0_{(j)}]_{n-1}$ to P_{king} where $[0_{(k)}]_{n-1}$ is used to generate $[0^{(i)}]_{n-1}$. Denote the coefficients to get $[0^{(i)}]_{n-1}$ (a row of the hyber-invertible matrix M) to be $\ell_k(1\le k\le n)$
5. P_{king} checks whether there exists P_j, P_k such that they send different values as the j^{th} sharing of $[0_{(k)}]_{n-1}$. If so, P_j and P_k are two disputed parties. Otherwise, there must exist j such that one of the followings holds:
   - the j^{th} sharing of $\overline{[x^{(i)}+a^{(i)}]}_{n-1} - [x^{(i)}+a^{(i)}]_t$ does not equal to the j^{th} sharing of $\sum_{k=1}^{n}\ell_k[0_{(k)}]_{n-1}$
   - $[0_{(j)}]_{n-1}$ is not a zero sharing

   Then P_j must be corrupted

It's easy to see that the above procedure allows an honest P_{king} to identify where the error comes from. To further ensure that a malicious P_{king} cannot charge two innocent parties, we need to make the following modification as the standard player elimination framework

- if P_{king} broadcasts that P_j is corrupted, all parties take P_{king}, P_j as two disputed parties who will be kicked out in the next round;

- if P_{king} broadcasts that P_j and P_k are inconsistent, P_{king} also broadcasts the index of the inconsistent message and the different values P_j and P_k sends. If P_j disagrees (he thinks the value is not what he sends to P_{king}), he broadcasts Disagree and all parties take P_{king}, P_j as two disputed parties. Similarly, if P_k broadcasts Disagree, all parties take P_{king}, P_k as two disputed parties. Otherwise, all parties take P_j, P_k as two disputed parties.

Remark 1. *The main difference between our protocol and the protocol in [GLS19] is the way of detecting the cheaters when the verification fails.*

They create redundant sharings called 4-Consistent tuples to help P_{king} recover the whole correct degree-$(n-1)$ Shamir secret sharing and then find the cheater.

Our key observation is that the randomness used in generating the random zero sharings is safe to reveal (after the broadcast is done) and is enough to help P_{king} find the cheater. Thus our simple protocol also works.

3 Preliminary

3.1 Player Elimination Framework

The player elimination framework, first introduced in [HMP00], is widely used to achieve perfect security in MPC protocols [BTH08, GLS19]. On a high level, we divide the computation that may have errors into several segments. The players first follow the computation within the segments and then verify the execution correctness. If there exists errors, at least one honest player will report the existence of errors. We call the player *unhappy* if he/she detects errors. If an error indeed happens, all players run another protocol to locate two players where at least one of them is corrupted. Then these two players are eliminated from the active player set $\mathcal{P}_A$ and this segment will be run again on the active player set. After some player elimination steps we have n' players where a maximum t' of them can be malicious. It follows that $n' - 2t' = n - 2t$, and we denote $T = n - 2t$. T remains constant throughout the protocol under our player elimination framework.

In the following we introduce the player elimination framework instance in [BTH08]. In the framework a procedure π is taken as an input output a procedure that either outputs the original output of π or outputs a pair of disputed parties to all parties. Each party maintains locally a *happy-bit* in the framework.

Procedure 1 (Player Elimination (π))**.**

1. **Initialize Phase**: All players have their happy-bits be `happy`.
2. **Computation Phase**: All players run π.
3. **Fault Detection Phase**:

- All players broadcast their happy-bits.
- For each party, if he or she receives at least one `unhappy` then sets his or her happy-bit to `unhappy`.
- All players run a consensus protocol on their happy-bits. If the consensus is `happy`, all parties output the output of π and terminate the procedure. Otherwise, they proceed to the following steps.

4. **Fault Localization Phase**:

   - All players agree the party with the smallest index in $\mathcal{P}_A$ as the dealer D. All other players send all their generated values and communication to D.
   - On receiving all the information, the dealer simulates π and the fault detection phase himself or herself. The dealer either prepares the message $(P_i, \texttt{Incorrect})$ if P_i failed to follow the protocol, or the message $(P_j, P_k, l, m, m', \texttt{Inconsistent})$ if the message m that P_j sends to P_k in round l is inconsistent with P_k's message m' that P_k claims to have received. Then D broadcasts the prepared message to all parties.
   - If $(P_i, \texttt{Incorrect})$ is broadcast, all players set the eliminating set $E = \{D, P_i\}$. Otherwise $(P_j, P_k, l, m, m', \texttt{Inconsistent})$ is broadcast. P_j and P_k will broadcast if they agree with D. If P_j does not agree, all players set the eliminating set $E = \{D, P_j\}$; if P_k does not agree, all players set the eliminating set $E = \{D, P_k\}$; otherwise all players set the eliminting set $E = \{P_j, P_k\}$.
   - All players update $\mathcal{P}_A := \mathcal{P}_A - E$ and terminate.

Suppose $\Omega(\pi)$ is the total communication cost in the procedure π. In the third step, the players communication a total of $O(n^2)$ bits. Then the communication cost for π with player elimination is $O(n^2 + \Omega(\pi))$ bits. The overhead is negligible asymptotically if $\Omega(\pi)$ is at least $O(n^2)$.

4 Protocol

In this section, we combine each parts to form our protocol. Only the way we multiply is different from Protocol 1.

Protocol 2. Our Protocol

1: Invoke the preparation protocol to get $2C$ Beaver triples $([a_k]_t, [b_k]_t, [c_k]_t)_{k=1}^{2C}$ where a_k, b_k are independent uniform random values and $c_k = a_k b_k$

2: We first deal with the input gates

1. For each input gate, we associate it with a triple $([a_k]_t, [b_k]_t, [c_k]_t)$
2. All parties send their shares of $[a_k]_t$ to the dealer who is the owner of the input
3. The dealer reconstructs a_k, and computes $d = x - a_k$ where x is the input
4. Invoke Broadcast to let every dealer robustly broadcast the differences d in batches efficiently
5. Each party computes $[x]_t = d + [a_k]_t$ as its input shares

3: Initialize $n' \leftarrow n, t' \leftarrow t$ and $\mathcal{P}' \leftarrow \mathcal{P}$ (the set of all parties)

4: All parties agree on a division of the remaining computation (except for the output gates) into t segments with roughly the equal length. Active parties (parties in $\mathcal{P}'$) evaluate each segment gate by gate as follows

- Before any computation, active parties generate $\lceil 2C/t \rceil$ degree-$(n'-1)$ random zero sharings by applying player elimination framework to the method we describe in Section 2.2.2. They agree on a party $P_{\text{king}} \in \mathcal{P}'$.
- Random Gate: Use the sharings $[a_k]_t$ in a fresh triple $([a_k]_t, [b_k]_t, [c_k]_t)$
- Addition Gate: Each party locally computes the addition of their shares
- Multiplication Gate: Given $[x]_t, [y]_t$, to compute $[xy]_t$
  1. Each active party consumes a new triple $([a]_t, [b]_t, [c]_t)$ and two fresh degree-$(n'-1)$ random zero sharings $[0^{(1)}]_{n'-1}, [0^{(2)}]_{n'-1}$
  2. Active parties locally compute $[x+a]_{n'-1} = [x]_t + [a]_t + [0^{(1)}]_{n'-1}$ and $[y+b]_{n'-1} = [y]_t + [b]_t + [0^{(2)}]_{n'-1}$, and send the shares to P_{king}
  3. P_{king} reconstructs $x+a, y+b$ and broadcasts the values to $\mathcal{P}'$
  4. Parties in $\mathcal{P}'$ compute $[xy]_t = (x+a)(y+b) - (x+a)[b]_t - (y+b)[a]_t + [c]_t$
- After completing all the computations of the segment, all the active parties do verification as follows
  1. Invoke CheckConsistency to check P_{king} indeed does the broadcast (instead of sending different values to different parties) in this segments. If the check fails, two disputed parties are kicked out from $\mathcal{P}'$, $n' \leftarrow n' - 2, t' \leftarrow t' - 1$ and all other parties will retry the computation of this segment;

2. Active parties locally compute $[x+a]_t = [x]_t + [a]_t$ and $[y+b]_t = [y]_t + [b]_t$, and invoke ReconsPubl to get the real values of $x+a, y+b$. If any party finds the real value is different from what P_{king} sends, he gets `unhappy`;
3. Active parties run a consensus protocol to see if someone of them is `unhappy`. If not, they take this segment's computation result as output, and move on to compute the next segment. Otherwise, suppose the first wrong value reconstructed by P_{king} is $x^{(i)} + a^{(i)}$, and all active parties do the fault localization as follows
   (a) All active parties send their shares of $[x^{(i)} + a^{(i)}]_t$ to P_{king}
   (b) P_{king} uses the shares to recover the correct shares $[x^{(i)} + a^{(i)}]_t$
   (c) P_{king} computes $[\overline{x^{(i)} + a^{(i)}}]_{n'-1} - [x^{(i)} + a^{(i)}]_t$, which is supposed to be a zero sharing $[0^{(i)}]_{n'-1}$
   (d) All active parties send the randomness they use in constructing $[0^{(i)}]_{n'-1}$ to P_{king}. Formally, for $j = 1, 2, \cdots, n'$, active party P_j sends his shares of $[0_{(k)}]_{n'-1} (1 \le k \le n')$ and the whole shares of $[0_{(j)}]_{n'-1}$ to P_{king} where $[0_{(k)}]_{n'-1}$ is used to generate $[0^{(i)}]_{n'-1}$. Denote the coefficients to get $[0^{(i)}]_{n'-1}$ (a row of the hyber-invertible matrix M) to be $\ell_k (1 \le k \le n')$
   (e) P_{king} checks whether there exists P_j, P_k such that they send different values as the j^{th} sharing of $[0_{(k)}]_{n'-1}$. If so, P_{king} broadcasts the index of the inconsistent message and the different values P_j and P_k send. Otherwise, P_{king} finds j such that one of the followings holds:
      – the j^{th} sharing of $[\overline{x^{(i)} + a^{(i)}}]_{n'-1} - [x^{(i)} + a^{(i)}]_t$ does not equal to the j^{th} sharing of $\sum_{k=1}^{n'} \ell_k [0_{(k)}]_{n'-1}$
      – $[0_{(j)}]_{n'-1}$ is not a zero sharing
      In this case, P_{king} broadcasts that P_j is corrupted.
   (f) If P_{king} broadcasts that P_j is corrupted, each party sets $E = \{P_j, P_{\text{king}}\}$
      If P_{king} broadcasts that P_j and P_k sends two different values, then P_j checks whether the value is what he sent to P_{king}. If not, P_j broadcasts Disagree, each party sets $E = \{P_j, P_{\text{king}}\}$. Similarly, if P_k broadcasts Disagree, each party sets $E = \{P_k, P_{\text{king}}\}$. Otherwise, each party sets $E = \{P_j, P_k\}$
   (g) $n' \leftarrow n' - 2, t' \leftarrow t' - 1, \mathcal{P}' \leftarrow \mathcal{P}' - E$. All the remaining active parties (parties in $\mathcal{P}'$) retry the computation of this segment.

5: For each output gate, the remaining active parties send his shares to the owner of the output, who reconstruct the output.

This protocol achieves $O(Cn\kappa + n^3\kappa)$ communication complexity as desired.

References

[Bea92] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology—CRYPTO'91: Proceedings 11*, pages 420–432. Springer, 1992.

[BTH08] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In *Theory of Cryptography: Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Proceedings 5*, pages 213–230. Springer, 2008.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19, 1988.

[DN07] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Advances in cryptology—CRYPTO 2007*, volume 4622 of *Lecture Notes in Comput. Sci.*, pages 572–590. Springer, Berlin, 2007.

[GBOW88] S Goldwasser, M Ben-Or, and A Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computing. In *Proc. of the 20th STOC*, pages 1–10, 1988.

[GLS19] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional mpc with guaranteed output delivery. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part II*, pages 85–114. Springer, 2019.

[GMW87] O Goldreich, S Micali, and A Wigderson. A completeness theorem for protocols with honest majority. In *STOC 87*, 1987.

[HMP00] Martin Hirt, Ueli Maurer, and Bartosz Przydatek. Efficient secure multi-party computation. In *Advances in Cryptology—ASIACRYPT 2000: 6th International Conference on the Theory and Application of Cryptology and Information Security Kyoto, Japan, December 3–7, 2000 Proceedings*, pages 143–161. Springer, 2000.

[Yao82] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

A Preparation Phase

A.1 Overview

In this section, we will show how [BTH08] generates the random triples. [BTH08] shows that if the triples are not secure, the honest parties can detect that. A further using of Player Elimination can give a bound of the total time.

A.2 Generating random shares

First, we introduce the DoubleShareRandom protocol. It either generates T independent secret values and their (d, d') secret sharing, or it fails. In the following context, when we refer to a matrix M, it is the hyper invertible matrix.

Procedure 2 (DoubleShareRandom (d, d'))**.**

1. **Secret Share**: Every $P_i \in \mathcal{P}'$ chooses a random s_i and acts (twice in parallel) as a dealer in Share to distribute the shares among the players in $\mathcal{P}'$, resulting in $[s_i]_{d,d'}$.

2. **Apply** M: The players in $\mathcal{P}'$ (locally) compute $\left([r_1]_{d,d'}, \ldots, [r_{n'}]_{d,d'}\right) = M\left([s_1]_{d,d'}, \ldots, [s_{n'}]_{d,d'}\right)$. In order to do so, every P_i computes his double-share of each r_j as linear combination of his double-shares of the s_k-values.

3. **Check**: For $i = T+1, \ldots, n'$, every $P_j \in \mathcal{P}'$ sends his double-share of $[s_i]_{d,d'}$ to P_i, who checks that all n' double-shares define a correct double-sharing of some value s_i. More precisely, P_i checks that all d-shares indeed lie on a polynomial $g(\cdot)$ of degree d, and that all d'-shares indeed lie on a polynomial $g'(\cdot)$ of degree d', and that $g(0) = g'(0)$. If any of the checks fails, P_i gets `unhappy`.

4. **Output**: The remaining T double-sharings $[r_1]_{d,d'}, \ldots, [r_T]_{d,d'}$ are outputted.

If all honest players are happy, then at least t' double-sharings are correct (the $n' - t'$ double-sharings inputted by honest players, as well as the t' double-sharings verified by honest players), and due to the hyper-invertibility of M, all $2t'$ double-sharings must be correct.

A.3 Generating triples

Then we use the following protocol to generate Beavers triples using the random sharings.

Procedure 3 (GenerateTriples)**.**

1. **Generate Double-Sharings**: Invoke DoubleShareRandom three times to generate the double-sharings $[a_1]_{t,t'}, \ldots, [a_T]_{t,t'}$, $[b_1]_{t,t'}, \ldots, [b_T]_{t,t'}$, and $[r_1]_{t,2t'}, \ldots, [r_T]_{t,2t'}$.
2. **Multiply**:
   - For $k = 1, \ldots, T$, the players in $\mathcal{P}'$ compute (locally) the $2t'$-sharing $[c_k]_{2t'}$ of $c_k = a_k b_k$ as $[c_k]_{2t'} = [a_k]_{t'} [b_k]_{t'}$ (by every player computing the product of his shares).
   - For $k = 1, \ldots, T$, the players in $\mathcal{P}'$ compute (locally) a $2t'$-sharing of the difference $[d_k]_{2t'} = [c_k]_{2t'} - [r_k]_{2t'}$.
   - Invoke `ReconsPubl` (Appendix B) to reconstruct $d_1, \ldots, d_T$ towards every player in $\mathcal{P}'$.
   - For $k = 1, \ldots, T$, the players in $\mathcal{P}'$ compute (locally) the t-sharing $[c_k]_t = [r_k]_t + [d_k]_0$, where $[d_k]_0$ denotes the constant sharing $[d_k]_0 = (d_k, \ldots, d_k)$.
3. **Output**: The t-shared triples $([a_1]_t, [b_1]_t, [c_1]_t), \ldots, ([a_T]_t, [b_T]_t, [c_T]_t)$ are outputted.

The security comes from the security of DoubleShareRandom. The total communication cost is $O(n^2\kappa)$.

B Batched Secret Reconstruction

Following the idea of the BH protocol [BTH08], there exist two protocols to reconstruct the Shamir sharings over some field of size $O(n)$. We note that these two protocols follow the player elimination framework. Denote t as the remaining number of corrupted players, n as the remaining number of players and d as the degree of the sharing to be reconstructed.

One reconstruction protocol is private where one player receives the shares from all parties and reconstruct the secret or becomes `unhappy`. We state this protocol as follows.

Protocol 3. ReconsPriv$(P, d, [s]_d)$

1. Each player sends his/her share of $[s]_d$ to player P.
2. If at least $d + t' + 1$ of the shares lies on a degree-d polynomial, P reconstructs the secret; otherwise P gets `unhappy`.

The total communication cost of ReconsPriv is $O(n)$ field elements, which is $O(n \cdot \log n)$ bits. We point out that following fact holds for the ReconsPriv protocol [BTH08]:

Lemma 1. *If $d < n' - 2t'$, then the player P can robustly recover the secret s from $[s]_d$. If $d < n' - t'$, then the player P can either correctly reconstruct the secret s or detect errors and get* ***unhappy****.*

The other reconstruction protocol is public where all players jointly recover T degree-d Shamir sharings $[s_1]_d, \cdots, [s_T]_d$. The γ_j in the protocol are pre-agreed points used as linear error correction code.

Protocol 4. ReconsPubl$(d, [s_1]_d, \cdots, [s_T]_d)$.

1. All players locally compute
$$[w_j]_d = \sum_{l=1}^{T} \gamma_j^{l-1} \cdot [s_l]_d.$$
2. Each player P_i invokes ReconsPriv to reconstruct $[w_i]_d$.
3. Each player P_i broadcast w_i or `unhappy`.
4. Each player P_i will reconstruct $s_1, \cdots, s_T$ on receiving at least $T + t$ consistent values; otherwise P_i gets `unhappy`.

The communication cost for `ReconsPubl` is $O(n^2)$ field elements, which is $O(n^2 \log n)$ bits. We have the following property for `ReconsPubl` [BTH08]:

Lemma 2. *If $d < n' - 2t'$, then all the players can robustly recover the secrets $\{s_i\}_{i=1}^T$ from $\{[s_i]_d\}_{i=1}^T$. If $d < n' - t'$, then either the players can correctly reconstruct the secrets or at least one honest player detects errors and gets* ***unhappy****.*