Multiplicative Depth Independent & Efficient MPC in the Presence of Mixed Adversary

Achintya Desai[1], Shubham Raj[1], and Kannan Srinathan[1]

International Institute of Information Technology-Hyderabad, India

Abstract. An extensive research of MPC protocols in different adversarial settings over the past few years has led to various improvements in this domain. Goyal et al.[13] in their paper addressed the issue of an efficient MPC protocol in active adversarial setting by removing the dependency on multiplication depth D_m in the arithmetic circuit. This development was followed by Hirt et al.[15] which proposed an efficient MPC protocol tolerating mixed adversary with communication complexity of $O((c_i + c_m + c_o)nk + c_iBA(k) + D_m(n^3k + nBA(k)))$ bits, where D_m is the multiplicative depth of the circuit. Additionally, Hirt et al.[15], proposed an open problem to construct a protocol for the mixed adversarial setting, independent of the multiplicative depth D_m, with linear communication complexity. In this paper, we resolve this problem in the affirmative by providing an efficient perfectly secure-MPC protocol in the mixed adversarial setting independent of the multiplicative depth of the circuit.

Keywords: Multi-Party Computation · Efficiency · Mixed Adversary · Multiplicative Depth

1 Introduction

Consider a scenario with n parties who do not trust each other. However, they want to jointly compute a function without revealing their private inputs. This function is represented as a circuit over a finite field. In cryptography literature, this problem is known as secure multi-party computation (MPC)[18]. This problem has been extensively studied in the modern literature with several different models of computation.

Adversarial setting is one such model of computation which has been one of the most prominent variation of secure multi-party computation. The three major types corruption that can be induced in an adversarial setting are perpetrated by active(which can behave in arbitrary way), passive(which can eavesdrop) and fail-stop adversaries(which can crash during execution). There exists other adversaries such as mobile adversary, but that's irrelevant to the scope of this work. Adversarial setting refers to the type of corruption that a protocol can withstand maintaining its correctness, consistency and completeness. Every adversarial setting comes with reliable boundary conditions on the number of parties that can be corrupted in previous influential works. It is well established

that a perfectly secure protocol with n parties can handle not more than $\frac{n}{3} - 1$ active corruptions and $\frac{n}{2} - 1$ passive corruptions. These boundary conditions can be improved provided the security of the protocol is relaxed.

In terms of practicality, while the passive setting does offer us a better boundary condition on the allowed number of corruptions, it is quite impractical and wildly optimistic. Active adversarial setting, on the other hand, does prepare us for the worst but is highly pessimistic and leaves us with a boundary condition of at most $\frac{n}{3} - 1$ corruptions. The real world isn't either black or white but a spectrum of colours. Similarly, a set of parties can be a combination of different types of corruptions. On top of that, a party can crash during execution of a protocol in the network which is a very realistic scenario. To counter this lack of practicality, Fitzi et al.[9] tried reaching a middle ground and pointed out a trade-off between the adversarial capabilities and the allowed number of corruptions. Fitzi et al.[9] relaxed the number of active corruptions represented as t_a and accommodated passive and crash fault corruptions represented as t_p and t_f respectively to create a more realistic model of adversarial setting called as mixed adversarial setting. This hybrid setting paved way for a more realistic scenario that accounts for all types of aforementioned corruptions with a boundary condition of $3t_a + 2t_p + t_f < n$.

A parallel discussion that has been around for quite a while is on the efficiency of the MPC protocols. Goldwasser et al. [12] and Chaum et al. [5] presented the first protocols for MPC in pure setting and Fitzi et al.[9] presented the same for mixed setting but they were expensive and incurred a communication cost of $\Omega(n^6)$ field elements in the evaluation of one multiplication gate. Over the years, many challenges have been overcome leading to improvements in pure and mixed adversarial settings through the works of [7], [16], [8], [2], [3], [11], [13], [15] making the communication complexity per multiplication gates linear in terms of the number of parties.

Table 1. MPC variations

Reference Papers	**Adversarial Setting**	**Communication Complexity**	**Dependency on Multiplicative Depth**
Beerliová-Trubíniová et al.[2]	Active	linear	✓
Goyal et al.[13]	Active	linear	✗
Hirt et al.[15]	Mixed	linear	✓
Our Paper	Mixed	linear	✗

Hirt et al. in their ITC 2020 paper [15] studied the efficiency of a secure MPC protocol in mixed adversarial setting. Their construction, however, yielded a protocol whose final communication complexity depended on the multiplicative depth of the circuit. In this paper, we answer the following open problem raised

by Hirt et al.[15] in affirmative: "*Is it possible to remove the dependency on the multiplicative depth of the circuit in the overall communication complexity of a secure MPC protocol while maintaining its efficiency and the restrictions on the number of corruptions such that* $3t_a + 2t_p + t_f < n$*?*"

For our construction, we assume that every pair of parties is connected by a secure channel and the communication between parties is synchronous.

2 Preliminaries

In our constructions, we will frequently use a few symbols to simplify the descriptions associated with them. The parties which are involved in the protocol are represented as a set P such that $P = \{P_1, ..., P_n\}$ is a set of n parties. The parties involved wants to compute a function that is represented as a circuit over a finite field $\mathcal{F}$. Circuits consist of various types of gates which frequently appears in the descriptions as well as the protocol complexities. Hence, using symbols to quantify different gates simplifies the representation of of the protocol. In our construction, we will use c_i input, c_o output, c_m multiplication, c_a addition and c_r random gates. This work also requires understanding of Byzantine agreement[10], Circuit randomization[1] and Hyper Invertible matrix[2] as a prerequisite whose extensive study material can be found in [6].

2.1 Adversary Model

Mixed Adversary: A **mixed adversary** is a more practical and realistic form of adversary that can corrupt the participating parties in Byzantine and fail-stop fashion. It is represented as (P, t_a, t_p, t_f) and can corrupt up to t_a, t_p and t_f parties in active, passive and fail-stop manner, respectively. In an **active** corruption, the adversary gains complete control of a party with the ability to modify the the messages that are sent. **Passive** corruption, on the other hand, only allows an adversary to read the messages sent by a party. A **fail-stop** corruption is an intricate detail that makes the **mixed adversary setting** a much more realistic possibility. In a fail-stop corruption, an adversary forces a node to crash however the crashed node follows the protocol honestly until the time of crash. An interesting property of a fail-stop corrupt party is that the adversary cannot see the internal state i.e. the inputs or the messages processed by the party (unless they are simultaneously actively or passively corrupted). Adversarial models often associate parties with tags such as honest, dishonest, correct, incorrect e.t.c which might leave some room of subtle differences. One such subtlety is the difference between a correct and an honest party. A **correct party** is one which has not been actively corrupted but can be either passively corrupted or is prone to crash however an **honest party** is one which correctly follows the protocol.

2.2 Byzantine Agreement

Byzantine agreement refers to the concept of non faulty nodes reaching an agreement upon a value in the presence of a Byzantine adversary. Formally, it can be represented as an output y taking a value x_i provided all the non faulty nodes agree upon a common value i.e. x_i through a consensus protocol. The idea of Byzantine consensus was initially conceived as interactive consistency amongst the nodes such that all non faulty nodes have the same value at the end which is ultimately the output. Our construction is based on mixed setting and Garay and Perry [10] formalized the guarantees of consensus in the mixed setting which include consistency and persistence. **Consistency** makes sure that the output of all parties that are not actively corrupted are equal and **persistence** makes sure that the input and output of all correct parties are same.
Broadcast is closely related to consensus such that in a broadcast protocol, every party agrees upon a value sent by a designated sender. Just like consensus, broadcast protocol too comes in various shapes and forms. A **reliable broadcast**, which is valid, is one in which a correct party sends a message and all other correct parties eventually receives it. A **terminating reliable broadcast** is a reliable broadcast that takes termination into account such that every correct party delivers some value.
In the mixed setting, we require the same consistency as in consensus. Garay and Perry [10] gave a consensus protocol for active setting with fail-stop corruption to achieve one bit consensus. This protocol allowed no passive corruptions such that $t_p = 0$ assuming that $3t_a + t_f < n$. The communication complexity of the protocol in [10] is $O(n^3)$. However, by applying the king-simulation technique of [4], this complexity can be reduced to $O(n^2)$.
To accommodate the mixed adversarial setting, we use a player-elimination framework to eliminate faulty parties thereby creating a reduced party set $\mathcal{P}'$. Byzantine consensus is invoked amongst parties in $\mathcal{P}'$. So, the following protocol is implemented which internally invokes a consensus protocol which is secure under the assumption that $3t_a + 2t_p + t_f < n$.

Protocol 1: Consensus$(\{x_i\}_{P_i \in \mathcal{P}'})$

- Consensus protocols in [10], [4] can be used by parties in $\mathcal{P}'$ such that the input of P_i is x_i.
- Each P_i sends its output to all parties in $\mathcal{P} \backslash \mathcal{P}'$ where $P_i \in \mathcal{P}'$.
- All parties $\in \mathcal{P}'$ outputs the result of the consensus, and all parties in $\mathcal{P} \backslash \mathcal{P}'$ determines its output using the rule of the majority.

The cost of Consensus in [10], [4] is $O(n^2)$ where n is the size of the initial set $\mathcal{P}$. Let $BA(k)$ be the complexity of broadcasting or reaching consensus on a $k-bit$ message in mixed setting. Hence using the aforementioned protocols gives us $BA(k) = O(n^2 k)$.

2.3 Circuit Randomization

Circuit randomization technique by Beaver et al.[1] helps in randomizing the inputs of a circuit and facilitates the pre-processing phase which reduces the overall communication complexity of the protocol. Given $z = xy$ such that x is shared as $[x]_d$ and y is shared as $[y]_d$ with d being the degree of polynomial used for sharing the secret, circuit randomization allows us to compute a sharing $[z]_d$ at the costs of two public reconstructions provided a random triple $[a]_d$, $[b]_d$, $[c]_d$ such that $c = ab$ which has been pre-shared is available. In this technique, we first prepare c_m shared multiplication triples $[a]_d$, $[b]_d$, $[c]_d$ and then we evaluate a circuit with c_m multiplications through a sequence of public reconstructions. In circuit randomization we express $z = xy$ as $z = ((x - a) + a)(y - b) + b)$. Let $d = x - a$ and $e = y - b$, then, $z = de + db + ae + c$, where (a, b, c) is a multiplication triple. If the multiplication triple (a, b, c) is random then d and e are random values irrespective of x and y. Hence, a sharing $[z]_d$ can be linearly computed as $[z]_d = [de]_d + d[b]_d + e[a]_d + [c]_d$, by reconstructing $[d]_d = [x]_d - [a]_d$ and $[e]_d = [y]_d - [b]_d$.

2.4 Secret Sharing

In this section, we are establishing secret sharing variations, similar to [15], as a fundamental building block for further protocols. We use Shamir's secret sharing scheme[17] to share and reconstruct secrets amongst n parties. In this work, *Share* protocol simply distributes the secret input of a player P_i using a random polynomial. *PrivReconRobust* protocol is a robust private reconstruction where secret is reconstructed towards a single party. *PubReconRobust* protocol reconstructs l secret sharings towards all the parties robustly with the help of player-elimination framework.

Share

Protocol 2: Share$(P_i, d, s, (P', t'_a, t'_p, t'_f))$

– Party P_i chooses a random polynomial g of degree d such that $s = g(0)$ and distributes share $s_j = g(\alpha_j)$ to party $P_j \in P'$.

Public Reconstruction

Protocol 3: PubReconRobust$(d, l, [s^1]_d, ..., [s^l]_d, (P', t'_a, t'_p, t'_f))$

For each bucket $[\hat{s}^1]_d, ..., [\hat{s}^l]_d$ of size $l \leq n' - t'_a$ do the following step 1 to 4 :

1. Expansion : All the parties compute locally $([u^1]_d, ..., [u^{n'}]_d)^T = V([\hat{s}^1]_d, ..., [\hat{s}^l]_d)^T$ where V is the Vandermonde matrix of size $n' \times l$ defined by fixed vector β with unique values.
2. Distributing the shares : For $i \in \{1, ..., n'\}$, the parties send their share of $[u^i]_d$ towards P_i.

3. Reconstructing secret : Each $P_i \in P'$ sends u^i to every $P_j \in P'$.
4. Checking validity : Every $P_j \in P'$ checks whether there exists a polynomial g of degree at most $l-1$ such that all points $(\beta^1, u^1), ..., (\beta^{n'}, u^{n'})$ lie on g. If this is the case, P_i considers $\hat{s}^1, ..., \hat{s}^l$ as correct. Otherwise, P_j sets happy bit as unhappy.
5. Fault Detection : All parties now perform fault detection step from player elimination framework. If the output is "happy" then the reconstructed values are considered correct and move to the next segment. Otherwise, continue to next step.
6. Fault localization : Let $P_r \in P'$ be the party with the smallest index. All parties $P_i \in P'$ sends to P_r the values generated, sent and received in the above steps. Also, message M_i received during Fault Detection step.
   Now, P_r simulates the above steps on the behalf of each $P_i \in P'$ to reconstruct the sharing polynomial and the correct shares of each party. If P_r identifies a P_i which does not follow the steps, then it broadcasts $(P_i, \textit{corrupt})$. If P_i finds a conflict between message index l where P_i should have sent x to P_k but P_k claimed to have received x' such that $x \neq x'$ then P_r broadcasts $(l, P_i, P_k, x, x', \textit{disputed})$.
   In case of dispute message broadcast by P_r, the conflicting parties P_i and P_k respond by broadcasting their stand (agree/disagree).
   If P_i disagrees, all parties set $E = \{P_i, P_r\}$. If P_k disagrees, all parties set $E = \{P_k, P_r\}$. Otherwise, every party sets $E = \{P_i, P_k\}$.
7. Crash-check fault localization : For party $P_h \in \{P_i, P_k, P_r\}$, the parties perform heartbeat protocol.
8. Player elimination : If all the parties P_i, P_r and P_k are alive, then all parties set $P' = P' \backslash E, n' = n' - 2$ and $t'_a = t'_a - 1$. Otherwise set E as parties for which were detected as crashed by heartbeat protocol. Set $P' = P' \backslash E, n' = n' - |E|$ and $t'_f = t'_f - |E|$. Repeat the procedure with updated (P', t'_a, t'_p, t'_f).

This robust protocol publicly reconstruct the secret under the assumption that $d < n - 2t'_a - t'_f$.

Robust Private Reconstruction

Protocol 4: PrivReconRobust$(P_i, d, [s]_d, (P', t'_a, t'_p, t'_f))$

1. Every party $P_j \in P'$ send their share s_j of the sharing $[s]_d$ to party P_i.
2. If there exists a polynomial g with party P_i of degree d such that atleast $d + t'_a + 1$ shares received must lie on g, then output $s = g(0)$. Otherwise P_i becomes unhappy.

This robust protocol privately reconstruct the secret towards a party under the assumption that $d < n - 2t'_a - t'_f$.

2.5 4-Consistency

[13] used 4-consistency to allow verification of randomized shares. In a protocol, we use random $(n'-1)$-shares in order to evaluate the circuit. We need all the shares from P' to reconstruct the value and there are no redundancies. However, due to the lack of redundancy, sharing becomes vulnerable which means the verification becomes harder. This means a party can change the value by changing its own share without being detected. To solve this issue, we need 4-consistency which allows each party to commit their shares after evaluation of the circuit thereby helping in the verification process. In our definition, n' is the number of active parties and t_a is the maximum number of actively corrupted parties an adversary can control.

Definition 1. *For a partition π of $P' = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3$ such that $|\mathcal{P}_1|, |\mathcal{P}_2|, |\mathcal{P}_3| \leq t_a + 1$, a tuple of t_a-sharings $[\![r]\!] = ([_0r]_{t_a}, [_1r]_{t_a}, [_2r]_{t_a}, [_3r]_{t_a})$ is a 4-consistent tuple with respect to π if ${}_0r = r$ and there exists a degree-$(n'-1)$ polynomial p() with p(0) = r and for all $P_i \in \mathcal{P}_j, p(\alpha_i)$ is the i^{th} share of the sharing $[_jr]_{t_a}$. [13]*

We generate $[r]_{t_a,n'-1}$ for evaluation. The terms $([_1r]_{t_a}, [_2r]_{t_a}, [_3r]_{t_a})$ are generated to commit the shares of $[r]_{n'-1}$ in verification.

Lemma 1. *4-consistency is preserved under linear combinations*

Proof. Refer to [13].

3 Preparation Phase

3.1 Generating Random Triple-Sharings

Identical to [13], this protocol *TripleShareRandom* generates l random triple sharings $[r]_{d,d',d''}$ and distributes them using polynomials of degree d, d', d''. Initially, all n' parties distribute their randomness using polynomials of degree d, d', d''. Hyper-invertible matrix, as described in [2], is applied which generates new sharings such that any $n' - t'_a$ output sharings are uniform random if any $n' - t'_a$ input sharings were random. The correctness of the output sharings is verified in step 3 by reconstructing $2t'_a$ sharings. The remaining l sharings are considered as output of the protocol.

Protocol 1: TripleShareRandom$(d, d', d'', l, (P', t'_a, t'_p, t'_f))$

1. Distributing randomness : Every party $P_i \in P'$ randomly chooses $s_i \in F$ and performs share protocol thrice among P' as Share$(P_i, d, s^i, (P', t'_a, t'_p, t'_f))$, Share$(P_i, d', s^i, (P', t'_a, t'_p, t'_f))$ and Share$(P_i, d'', s^i, (P', t'_a, t'_p, t'_f))$. Each party in $P_j \in P'$ receives j-th share of $[s^i]_{d,d',d''}$.
2. Applying hyper-invertible matrix : Every party computes locally

$[r^1]_{d,d',d''}, [r^2]_{d,d',d''}, ..., [r^{n'}]_{d,d',d''} =$
$M([s^1]_{d,d',d''}, [s^2]_{d,d',d''}, ..., [s^{n'}]_{d,d',d''})$ where M is hyper-invertible matrix of size $n' \times n'$.

3. Checking correctness : All parties $P_i \in P'$ send their shares of $[r^j]_{d,d',d''}$ to respective P_j where $j \in \{1, 2, ..., 2t'_a\}$. Each P_j with $j \in \{1, 2, ..., 2t'_a\}$ checks whether the received triple sharing is correct or not by constructing polynomial g_1 with degree d from $[r^j]_d$, polynomial g_2 with degree d' from $[r^j]_{d'}$ and polynomial g_3 with degree d'' from $[r^j]_{d''}$. If $g_1(0) = g_2(0) = g_3(0)$ does not hold then P_j sets its happy-bit as unhappy.
4. Output : All parties in P' consider remaining l sharings as output i.e. $[r^{n'-l+1}]_{d,d',d''}, ..., [r^{n'}]_{d,d',d''}$.

Lemma 2. ***TripleShareRandom*** *detectably generates l correct (d, d', d'') - sharings and the shared value corresponding to each sharing is uniformly random in the presence of adversary provided that (P', t'_a, t'_p, t'_f) is valid and input values of parties are consistent. Also, the communication complexity of* ***TripleShareRandom*** *is $O(ln'k + n'^2k)$*

Proof. We prove the correctness and security with the assumption that no party crashed during the protocol execution and all correct parties are happy at the end of the protocol.
Correctness : The values $s^i \in F$ randomly chosen by parties $P_i \in P'$ in step 1 of the protocol, are generated & shared properly by atleast $n' - t'_a$ parties which makes at least $n' - t'_a$ sharings from $[s^i]_{d,d',d''}$ to be correct. In step 3 of the given protocol, at least t'_a sharings of $[r^i]_{d,d',d''}$ are verified by correct parties. In total, there are n' correct sharings ($n' - t'_a$ sharings of $[s^i]_{d,d',d''}$ and t'_a sharings of $[r^i]_{d,d',d''}$). Due to the bijective property of hyper-invertible matrix, any other sharing can be written as a linear combination of the correct n' sharing which makes all the involved sharings correct.
Privacy : $t'_a + t'_p$ values of s^i(Shares generated by actively or passively corrupted parties) and $\min(2t'_a, t'_a + t'_p)$ values of r^i (reconstructed in verification phase) are known to the adversary. In total, $2t'_a + t'_p + \min(t'_a, t'_p)$ values are fixed by adversary. Hence, there exists a bijective mapping between the last $l \leq n' - 2t'_a - t'_p - \min(t'_a, t'_p)$ values of r^i and l values of s^i generated by either honest or fail-stop corrupted parties. Hence, the values $[r^{n'-l+1}]_{d,d',d''}, ..., [r^{n'}]_{d,d',d''}$ outputted by the protocol are uniformly random and hidden from the adversary.
Complexity : Each execution of the above protocol results in $O(n'^2k)$ bits of communication(n' shares are broadcasted to n' parties). Each execution generates $l \leq n' - 2t'_a - t'_p - \min(t'_a, t'_p)$ double-sharings. Since $3t'_a < n'$ and $2t'_p < n'$, $l \leq n' - 2t'_a - t'_p - \min(t'_a, t'_p) = \frac{1}{5}n'$. Therefore, the complexity of TripleShareRandom is $O((\frac{5l}{n'} + 1)n'^2k) = O(ln'k + n'^2k)$

3.2 Generating Random Multiplication Tuples

Goyal et al. [13] described the steps to generate multiplication tuples, which we have inherited in this work. Initially, all the parties in P' execute TripleShareRandom protocol to generate and distribute three random values a, b and r. The values a and b, are distributed using polynomials of degree d, d' and $n'-1$ whereas r is distributed using polynomials of degree $d, 2d'$ and $n'-1$. Now, each party locally computes $[c]_{2d'} = [a]_{d'} \times [b]_{d'}$ and $[e]_{2d'} = [c]_{2d'} - [r]_{2d'}$. In the following step, the parties publicly reconstruct e towards all parties. Then each party locally compute their d-sharing of product as $[r]_d + e$. At the end, l multiplication triplets (a, b, c) are formed and all parties hold their respective sharings along polynomial d and $n'-1$.

Protocol 2: GenerateMultiplicationTriples$(d, l, (P', t'_a, t'_p, t'_f))$

1. Generate three random triples : All parties in $P_i \in P'$ invoke *TripleShareRandom*$(d, d', n'-1, l, (P', t'_a, t'_p, t'_f))$, *TripleShareRandom*$(d, d', n'-1, l, (P', t'_a, t'_p, t'_f))$ and *TripleShareRandom*$(d, 2d', n'-1, l, (P', t'_a, t'_p, t'_f))$, where $d' = t'_a + t'_p$, in parallel to generate random triple sharings for $[a^1]_{d,d',n'-1}, ..., [a^l]_{d,d',n'-1}$, $[b^1]_{d,d',n'-1}, ..., [b^l]_{d,d',n'-1}$ and $[r^1]_{d,2d',n'-1}, ..., [r^l]_{d,2d',n'-1}$ respectively.
2. Local Computations : All parties $P_i \in P$ such that $i \in \{1, ..., l\}$, locally compute $[c^i]_{2d'} = [a^i]_{d'}.[b^i]_{d'}$ and $[e^i]_{2d'} = [c^i]_{2d'} - [r^i]_{2d'}$
3. Reconstructing the blinded product : All parties perform *PubReconRobust* to reconstruct $(e^1, ..., e^l)$ towards all parties in P'
4. Forming l triplets : Parties $P_i \in P$ such that $i \in \{1, ..., l\}$ locally compute d-sharing of $c^i = a^i.b^i$ as $[c^i]_d = [r^i]_d + e^i$
5. Output : Parties P_i where $i \in \{1, ..., l\}$ output l triplets as $([a^1]_{d,n'-1}, [b^1]_{d,n'-1}, [c^1]_{d,n'-1}), ..., ([a^l]_{d,n'-1}, [b^l]_{d,n'-1}, [c^l]_{d,n'-1})$

Lemma 3. ***GenerateMultiplicationTriples*** *detectably generates l correct triples of d-sharings and the shared values a and b corresponding to each triplet sharing $([a]_d, [b]_d, [c]_d)$ are uniformly random in the presence of adversary provided that (P', t'_a, t'_p, t'_f) is valid and input values of parties are consistent. Also, the communication complexity of* ***GenerateMultiplicationTriples*** *is $O(ln'k + n'^2k)$*

Proof. Correctness and Secrecy : Correctness and secrecy follows from lemma *TripleShareRandomlemma* and *PubReconRobust* since the degree, $2d'$, in reconstructing the blinded product phase satisfies the condition $2d' = 2t'_a + 2t'_p < n' - t'_a$. Complexity : Since the protocol is based on $TripleShareRandom$ and *PubReconRobust*, the complexity $O(ln'k + n'^2k)$ follows from lemma 3.1 and *PubReconRobust*.

4 Constructions

4.1 Generating 4-Consistent Tuples

This protocol is obtained from [13] which generates l correct and random 4-consistent tuples $[\![r]\!] = ([_0r]_d, [_1r]_d, [_2r]_d, [_3r]_d)$. $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ are partitions of parties such that $\mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 = P$ and $|\mathcal{P}_1|, |\mathcal{P}_2|, |\mathcal{P}_3| \leq d+1$. The partitions allows to create redundancy of matrix $([s^1], ..., [s^n]_{n-1})$ such that the entire matrix can be recovered even if the adversary corrupts its shares. Each row of this matrix is chosen d sharing such that even if adversary tampers its shares, the original matrix and the 3 matrix which belong to each partition can be recovered. For extensive details, refer to [13]. Each party distributes a random 4-consistent tuple, and using hyper-invertible matrix new random sharings are constructed. For verifying the correctness, $2t'_a$ 4-consistent tuples are reconstructed. If they are invalid, parties set their happy bit as unhappy. Otherwise l 4-consistent tuples are outputted.

Protocol 1: QuadrupleShareRandom$(l, (P', t'_a, t'_p, t'_f), (\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3))$

1. Distributing randomness : Every party $P_i \in P'$ produces a random 4-consistent tuple $[\![s^i]\!] = ([_0s^i]_d, [_1s^i]_d, [_2s^i]_d, [_3s^i]_d)$. Each party in P_j receives j-th share of $[\![s^i]\!]$ from P_i.
2. Applying hyper-invertible matrix : Every party computes locally $([\![r^1]\!], [\![r^2]\!], ..., [\![r^{n'}]\!]) = M([\![s^1]\!], [\![s^2]\!], ..., [\![s^{n'}]\!])$ where M is hyper-invertible matrix of size $n' \times n'$.
3. Checking correctness : All parties $P_i \in P'$ send their shares of $[\![r^j]\!]$ to respective P_j where $j \in \{1, 2, ..., 2t'_a\}$. Each P_j with $j \in \{1, 2, ..., 2t'_a\}$ checks whether the received 4-consistent tuple $[\![r^j]\!]$ is correct or not. If it is invalid then P_j sets its happy-bit as unhappy.
4. Output : All parties in P' consider remaining l sharings as output i.e. $[\![r^{n'-l+1}]\!], ..., [\![r^{n'}]\!]$.

Lemma 4. ***QuadrupleShareRandom*** *detectably generates l correct random 4-consistent tuple and the shared value corresponding to each tuple sharing $([_0r]_d, [_1r]_d, [_2r]_d, [_3r]_d)$ are uniformly random in the presence of adversary provided that (P', t'_a, t'_p, t'_f) is valid and input values of parties are consistent. Also, the communication complexity of* ***QuadrupleShareRandom*** *is $O(ln'k + n'^2k)$*

Proof. Correctness and Secrecy : Since 4 sharings are distributed instead of 3, the correctness and secrecy follows from lemma 3.1.
Complexity : Since the protocol is similar to $TripleShareRandom$, the complexity $O(ln'k + n'^2k)$ follows from lemma 3.1.

4.2 Generating Random 0-Sharings

Description ZeroShareRandom is yet another protocol that was described by Goyal et al. in [13] which has been inherited in this work. This protocol

generates l random t-sharings of 0. Initially, parties generates l random triple sharings of r using a polynomial of degree d', such that $([a]_{d'}, [b]_{d'}, [c]_{d'})$ are the triple sharings and $r = a = b = c$. The parties then locally compute $[b]_{d'} - [a]_{d'}$ to finally evaluate their shares of 0. *ZeroShareRandom* is used for ouptut gate evaluation.

Protocol 2: ZeroShareRandom$(l, (P', t'_a, t'_p, t'_f))$

1. Distributing random triple sharings : Every party $P_i \in P'$ performs *ThreeShareRandom*$(d', d', d', l, (P', t'_a, t'_p, t'_f))$ where $d' = t'_a + t'_p$ to generate $[r^{n'-l+1}]_{d',d',d'}, ..., [r^{n'}]_{d',d',d'}$. If *ThreeShareRandom*$(d', d', d', l, (P', t'_a, t'_p, t'_f))$ outputs disputed parties then all parties halt.
2. Local computation : Every triple sharing $([r^j]_{d',d',d'})$ for $j \in \{n' - l + 1, ..., n\}$ is expressed as $([a^j]_{d'}, [b^j]_{d'}, [c^j]_{d'},)$ where the value is the same i.e. $r^j = a^j = b^j = c^j$. Every party locally computes $[b^j]_{d'} - [a^j]_{d'}$ for $j \in \{n' - l + 1, ..., n\}$ to obtain shares of 0.
3. Output : All parties consider $[0^{n'-l+1,...,n'}]_{d'}, ..., [0^{n'}]_{d'}$ as output.

Lemma 5. ***ZeroShareRandom*** *detectably generates l correct t - sharings of 0 and the share value corresponding to each sharing is uniformly random in the presence of adversary provided that (P', t'_a, t'_p, t'_f) is valid and input values of parties are consistent. Also, the communication complexity of* ***ZeroShareRandom*** *is $O(ln'k + n'^2k)$*

Proof. Correctness and Secrecy : Since *ZeroShareRandom* invokes *TripleShareRandom*, the correctness and secrecy follows from lemma 3.1. Complexity : Since the protocol is based on $TripleShareRandom$, the complexity $O(ln'k + n'^2k)$ follows from lemma 3.1.

4.3 Checking Consistency

Like [13] version of *CheckConsistencyKing*, it is used to check whether P_{King} is corrupt or not during the execution of *AdditionMultiplicationGateEval*. In the first step, given l sharings are extended into $l + d'$ using hyper-invertible matrix. These shares are given to each party and it checks whether it is valid or not. If it is invalid then it sets its happy bit as unhappy indicating P_{King} might be corrupt. Further, fault detection and localization steps detect, using a referee party P_r, which pair of parties cheated during the protocol. Also, if any of the parties crashed during the execution of the protocol then these are added to set of crashed parties.

Protocol 3: CheckConsistencyKing$(l, P_{king}, (P', t'_a, t'_p, t'_f), [d^1]_0, ..., [d^l]_0)$

Let $d' = t'_a + t'_p$.

1. Applying hyper-invertible matrix : Every party computes locally $[r^1]_0, [r^2]_0, ..., [r^{l+d'}]_0 = M([d^1]_0, [d^2]_0, ..., [d^l]_0)$ where M is hyper-invertible matrix of size $(l + d') \times l$.
2. Distributing Shares : All parties send the shares of $[r^j]_0$ to P_j where $j \in \{1, ..., l + d'\}$
3. Checking validity : If all shares of $[r^j]_0$ are equal then $[r^j]_0$ is valid. Otherwise P_j sets its happy-bit as unhappy.
4. Fault detection : Each party sends its own happy-bit to every other party.
   A party sets its own happy-bit as unhappy if it receives unhappy bit from at least one party or it does not receive any bit from at least one party.
   Perform consensus protocol on the modified happy bits. If the outcome is happy then all parties halt. Otherwise, perform next step.
5. Fault localization : Let $P_r \in P'$ be the party with the smallest index. All parties $P_i \in P_r$ sends to P_r the values generated, sent and received in the above steps. Also, P_{king} sends $d^1, ..., d^l$ to P_r.
   Now, P_r simulates the above steps on the behalf of each $P_i \in P'$ to check the correctness of the result generated. If P_r identifies a P_i which does not follow the steps, then it broadcasts $(P_i, corrupt)$. If P_i finds a conflict between message index l where P_i should have sent x to P_k but P_k claimed to have received x' such that $x \neq x'$ then P_r broadcasts $(l, P_i, P_k, x, x', disputed)$.
   In case of dispute message broadcast by P_r, the conflicting parties P_i and P_k respond by broadcasting their stand (agree/disagree).
   If P_i disagrees, all parties set $E = \{P_i, P_r\}$. If P_k disagrees, all parties set $E = \{P_k, P_r\}$. Otherwise, every party sets $E = \{P_i, P_k\}$.
6. Crash-check fault localization : For party $P_h \in \{P_i, P_k, P_r\}$, the parties perform heartbeat protocol.
7. Output : If all the parties P_i, P_r and P_k are alive, then all parties in P' consider E as output. Otherwise set E as parties for which were detected as crashed by heartbeat protocol.

Lemma 6. ***CheckConsistencyKing*** *securely checks if a party P_{king} sent l same elements to all other parties in the presence of mixed adversary provided that (P', t'_a, t'_p, t'_f) is valid. Also, the communication complexity of* ***CheckConsistencyKing*** *is $O(nk + (t_a + t_f)(n^2k + BA(k)))$*

Proof. Complexity : In the distributing shares step, all parties send $O(n^2)$ elements. The rest of the steps are similar to party elimination framework except P_{king} sends additional $O(n)$ elements to P_r during fault-localization step. Hence, the overall communication complexity is $O(nk + (t_a + t_f)(n^2k + BA(k)))$. .

4.4 Checking 4-Consistent Tuples

Check4ConsistentTuple generates and checks whether each party distributed a correct random 4-consistent tuple. Firstly, each party generates l random 4-consistent tuples. It generates such $\lceil \frac{n'}{l} \rceil$ sets of l random tuples of 4-consistent sharings. Each random 4-consistent tuple is associated with corresponding input 4-consistent tuples. Instead of revealing the input 4-consistent tuples, all parties perform checking over the addition of the two tuples. Further, all parties apply hyper-invertible matrix to obtain $l + d$ random 4-consistent tuples. Each of these tuples is now reconstructed by different parties and it checks whether it is valid or not. If it is invalid then it sets its happy bit as unhappy implying that 4-consistent tuple has been modified. Further, fault detection and localization steps detect, using a referee party P_r, which pair of parties cheated during the protocol. Here, the referee party checks the summation of the 4-consistent tuples instead of the original input 4-consistent tuples. Also, if any of the parties crashed during the execution of the protocol then these are added to set of crashed parties.

Protocol 4: Check4ConsistentTuple$(l, (P', t'_a, t'_p, t'_f), (\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3), \{[\![s^j]\!]\}_{j=1}^{n'})$

Let $d' = t'_a + t'_p$.

1. 4-consistent tuple generation : All parties invoke *QuadrupleShareRandom* to generate l random 4-consistent tuples denoted as $\{[\![r^1]\!], ..., [\![r^l]\!]\}$.
   If the *QuadrupleShareRandom* outputs disputed party pair $(P_i, P_k, disputed)$, then all parties forward the same pair as output and halt.
   This step is repeated to generate $\lceil \frac{n'}{l} \rceil$ sets of l random tuples of 4-consistent sharings which are denoted as $\{[\![r^{lj+1}]\!], ..., [\![r^{lj+l}]\!]\}$ where $j \in \{0, ..., \lceil \frac{n'}{l} \rceil - 1\}$
2. Distributing Shares : All parties send the shares of $[\![r^j]\!]$ to P_j and compute $[\![u^j]\!] = [\![s^j]\!] + [\![r^j]\!]$ where $j \in \{1, ..., n'\}$
3. Applying hyper-invertible matrix : Every party computes locally $([\![v^1]\!], [\![v^2]\!], ..., [\![v^{l+d'}]\!]) = M([\![u^{lj+1}]\!], ..., [\![u^{lj+l}]\!])$ where M is hyper-invertible matrix of size $(l + d') \times l$.
   For $k \in \{1, ..., l + d'\}$, all parties send the shares of $[\![r^k]\!]$ to P_k.
   If the received 4-consistent tuple is valid then P_k sets its happy-bit as happy. Otherwise P_j sets its happy-bit as unhappy.
   Repeat this steps for values of j given as $j \in \{0, ..., \lceil \frac{n'}{l} \rceil - 1\}$.
4. Fault detection : Each party sends its own happy-bit to every other party.
   A party sets its own happy-bit as unhappy if it receives unhappy bit from at least one party or it does not receive any bit from at least one party.
   Perform consensus protocol on the modified happy bits. If the outcome is happy then all parties halt. Otherwise, perform next step.

5. Fault localization : Let $P_r \in P'$ be the party with the smallest index. All parties $P_i \in P_r$ sends to P_r the values generated, sent and received in the above steps. Also, each party P_i sends $[\![u^i]\!]$ to P_r.
   Now, P_r simulates the above steps on the behalf of each $P_i \in P'$ to check the correctness of the result generated. If P_r identifies a P_i which does not follow the steps, then it broadcasts $(P_i, \mathit{corrupt})$. If P_i finds a conflict between message index l where P_i should have sent x to P_k but P_k claimed to have received x' such that $x \neq x'$ then P_r broadcasts $(l, P_i, P_k, x, x', \mathit{disputed})$.
   In case of dispute message broadcast by P_r, the conflicting parties P_i and P_k respond by broadcasting their stand (agree/disagree).
   If P_i disagrees, all parties set $E = \{P_i, P_r\}$. If P_k disagrees, all parties set $E = \{P_k, P_r\}$. Otherwise, every party sets $E = \{P_i, P_k\}$.
6. Crash-check fault localization : For party $P_h \in \{P_i, P_k, P_r\}$, the parties perform heartbeat protocol.
7. Output : If all the parties P_i, P_r and P_k are alive, then all parties in P' consider E as output. Otherwise set E as parties for which were detected as crashed by heartbeat protocol.

Lemma 7. ***Check4ConsistentTuples*** *securely checks if each party distributed a correct 4-consistent tuple in the presence of mixed adversary provided that* (P', t'_a, t'_p, t'_f) *is valid. Also, the communication complexity of* ***Check4ConsistentTuples*** *is* $O(2nk + (t_a + t_f)(2n^2k + BA(k)))$

Proof. Complexity : In the distributing shares step, all parties send $O(n^2)$ elements to reconstruct n' number of $[\![r^j]\!]$s. Also, all parties send $O(n^2)$ more elements for reconstructing $l + t'$ number of $[\![v^k]\!]$s. The rest of the steps are similar to party elimination framework except each P_i sends additional elements to P_r during fault-localization step which makes up $O(n^2)$ elements. Hence, the overall communication complexity is
$O(n^2k + (t_a + t_f)(2n^2k + BA(k)))$.

5 Evaluation Phase

5.1 Input Gates

For each input gate, a party who holds the corresponding input performs this protocol. Initially, the initiating party performs private reconstruction of a random value r to itself. It then subtracts this random value from its input to obtain blinded input. This blinded input is broadcasted by the initiating party to all the other parties. Further, The initiating party performs *Heartbeat* protocol. If the output of *Heartbeat* is "alive", then each party computes its own share of input value by adding the blinded input with the random value and the protocol outputs "success". Otherwise the protocol outputs "fail" indicating faulty input party.

Protocol 1: InputGateEval$(P_i, [r]_d, (P', t'_a, t'_p, t'_f))$

Let s^k be the input of party P_i which wants to evaluate input gate.

1. Party P_i invokes *PrivReconRobust*$(P_i, d, [r]_d, (P', t'_a, t'_p, t'_f)$ to reconstruct r towards itself.
2. P_i broadcasts $e = s^k - r$ to all other parties.
3. P_i performs heartbeat protocol.
4. If the output of heartbeat protocol is "alive" then all parties output "success" and each party $P_j \in P'$ performs $e + r_j$, where r_j is P_j's share of r, to obtain share of $e + r$. Otherwise, each $P_j \in P'$ output "fail".
5. Repeat the above steps for $k \in \{1, ..., c_i\}$

Lemma 8. ***InputGateEval*** *securely distributes secret s_i for each party P_i in the presence of mixed adversary provided that (P', t'_a, t'_p, t'_f) is valid. Also, the communication complexity of* ***InputGateEval*** *is $O(c_i(nk + BA(k))$*

Proof. Correctness and Secrecy : Correctness and secrecy is followed by robust reconstruction of randomness towards P_i and blinded secret.
Complexity : From *PrivReconRobust*, $O(n)$ bits are communicated for reconstruction step. For c_i input gates, the overall complexity becomes $O(c_i(nk + BA(k))$

5.2 Random Gates

RandomGateEval, similar to Goyal et al. [13], uses *TripleShareRandom* along with player-elimination framework. If the output of *TripleShareRandom* is a pair of disputed parties or a set of crashed parties then such parties are eliminated and the aforementioned steps are repeated again. Otherwise, out of the d, d' and $n' - 1$ sharings, $d', n' - 1$ sharings are discarded and the l d-sharings for c_r gates are taken as output for all parties.

Protocol 2: RandomGateEval$(l, P_i, [r]_d, (P', t'_a, t'_p, t'_f))$

1. All Parties invokes *PlayerElimination(TripleShareRandom*$(d, d', n' - 1, l, (P', t'_a, t'_p, t'_f)))$ to construct random sharings.
2. If the output of *PlayerElimination(TripleShareRandom*$(d, d', n' - 1, l, (P', t'_a, t'_p, t'_f)))$ is a pair of disputed parties or set of crashed parties denoted as $P_{faultyset}$ then remove these parties from further computation such that $P' = P' \backslash P_{faultyset}$ and repeat the above step with modified P'.
3. Otherwise, discard the $(n' - 1)-$sharings and $d'-$sharings. The l number of $d-$sharings are denoted as $\{[\![[v^{lj+1}]_d]\!], ..., [\![[v^{lj+l}]_d]\!]\}$
4. Repeat the above steps for $j \in \{0, ..., \lceil \frac{c_r}{l} \rceil - 1\}$

Lemma 9. ***RandomGateEval*** *securely generates correct random d-sharings and the shared value corresponding to each sharing is uniformly random in the presence of mixed adversary provided that* (P', t'_a, t'_p, t'_f) *is valid. Also, the communication complexity of* ***RandomGateEval*** *is* $O((\lceil \frac{c_r}{l} \rceil + (t_a + t_p))(lnk + n^2k + BA(k)))$

Proof. Complexity : *TripleShareRandom* requires $O(lnk + n^2k + BA(k))$ which is repeated either $\lceil \frac{c_r}{l} \rceil$ times or $t_a + t_p$ times. Hence, the overall complexity becomes $O((\lceil \frac{c_r}{l} \rceil + (t_a + t_p))(lnk + n^2k + BA(k)))$

5.3 Addition and Multiplication Gates

A circuit is divided into multiples segments. For each segment *seg*, *AdditionMultiplicationGateEval* computes multiplication gates present. Unlike [15], there is no restriction on the depth of the multiplication gates present in a segment. Initially, partition P' is aggreed among the parties in the protocol. Protocol begins with the generation of multiplication triple per multiplication gate. Using circuit randomization technique from [1] along with P_{King}, all parties evaluate multiplication gates in the segment. Similarly, addition gate is evaluated with each party adding their shares locally of the respective inputs. Since, $n' - 1$-sharings are used during the computation of multiplication gates, it prohibits P_{King} from learning any additional information. To detect whether incorrect shares have been sent to P_{King} or P_{King} is corrupt during multiplication gate evaluation, all parties perform *CheckConsistencyKing*. Once it is confirmed that the values sent by P_{King} are not tampered, for verifying that the reconstructed values were correct parties perform public reconstruction using d-sharings. If the reconstructed values are same as obtained during gate evaluation then the output of the evaluation in step 3 is considered correct. Otherwise, all parties verify the randomness used during the generation of multiplication triples by committing the randomness with the help of 4-consistent tuples. This 4-consistent tuple is now verified by *Check4ConsistentTuple*. If the 4-consistent tuple is invalid then it returns pair of disputed or set of crashed parties. Otherwise, P_{King} identifies the disputed parties by finding mismatched value $[q]_d$ during gate evaluation phase and reconstruction using values from 4-consistent tuple.

Protocol 3: AdditionMultiplicationGateEval$(seg, (P', t'_a, t'_p, t'_f))$

Let $P_{king} \in P'$ be the party with highest index and $d = t_a + t_p$.

1. Initialization : All parties agree upon a partition $P' = \mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3$ such that $|\mathcal{P}_1|, |\mathcal{P}_2|, |\mathcal{P}_3| \leq d + 1$
2. Generate Multiplication Triples : All parties perform *PlayerElimination(GenerateMultiplicationTriples*$(d, d', n' - 1, l, (P', t'_a, t'_p, t'_f))$*)*.

If the output of *PlayerElimination(GenerateMultiplicationTriples*$(d, d', n'-1, l, (P', t'_a, t'_p, t'_f)))$ is a pair of disputed parties or set of crashed parties denoted as $P_{faultyset}$ then all parties take it as output and halt. Otherwise, continue to the next step with one multiplication triple corresponding to each multiplication gate in the segment *seg*.

3. Evaluate gates : For each addition gate in segment *seg*, all parties perform addition over their own shares.
   For each multiplication gate in segment *seg*, we denote input sharings as $[x]_d, [y]_d$ and corresponding multiplication triple as $([a]_{d,n'-1}, [b]_{d,n'-1}, [c]_{d,n'-1})$.
   All parties compute $[q]_{n'-1} = [x]_d + [a]_{n'-1}$ and $[e]_{n'-1} = [y]_d + [b]_{n'-1}$.
   All parties send the shares of $[q]_{n'-1}$ and $[e]_{n'-1}$ to P_{king}.
   P_{king} reconstructs values of q and e which are broadcasted to all other parties.
   All parties now compute output sharings as $[z]_d = qe - q[b]_d - e[a]_d + [c]_d$.
4. Validate consistency for P_{king} : We denote the elements distributed by P_{king} in previous step as $q^1, ..., q^l$ and $e^1, ..., e^l$ which is same as distributing $[q^1]_0, ..., [q^l]_0$ and $[e^1]_0, ..., [e^l]_0$.
   All parties perform *CheckConsistencyKing*$(l, P_{king}, (P', t'_a, t'_p, t'_f), [q^1]_0, ..., [q^l]_0)$ and *CheckConsistencyKing*$(l, P_{king}, (P', t'_a, t'_p, t'_f), [e^1]_0, ..., [e^l]_0)$. If the output from either of the two executions is a pair of disputed parties or set of crashed parties denoted as $P_{faultyset}$ then all parties take it as output and halt.
5. Rechecking the reconstructions : For each multiplication gate in *seg*, each party computes $[q]_d = [x]_d + [a]_d$ and $[e]_d = [y]_d + [b]_d$.
   For given segment *seg*, we denote the values of q and e as $q^1, ..., q^l$ and $e^1, ..., e^l$.
   All parties performs public reconstruction by invoking *PubReconRobust* with $[q^1], ..., [q^l]$ and $[e^1], ..., [e^l]$).
   Every party now checks whether the reconstructed values $q^1, ..., q^l$ and $e^1, ..., e^l$ match with the values distributed by P_{king} in step 3. If all values match then the output shares generated in step 3 are considered correct and taken as output for *seg*. Otherwise, continue to the next step with the first incorrect value be denoted as q^{i^*}.
6. Rechecking the randomness from multiplication triples of party P_i :
   Let $[s^i]_{d,n'-1}$ be the $d-$ sharing and $(n'-1)-$ sharing of s^i which a party P_i distributed during one of the invokation of *TripleShareRandom* during evaluation of multiplication triples in step 1.

Party P_i randomly generates $[_1s^i]_d, [_2s^i]_d, [_3s^i]_d$ such that k^{th} share of $[_js^i]$ is $[s^i]_{n'-1}$ for $j \in \{1,2,3\}$ and $P_k \in \mathcal{P}_j$. For $P_k \in P' \backslash \mathcal{P}_j$, P_i sends k^{th} share of $[_js^i]_d$ to P_k.

7. Check 4-Consistency : Let $[\![s^i]\!]$ denote the tuple sharings $[_0s^i]_d, [_1s^i]_d, [_2s^i]_d, [_3s^i]_d$ where $[_0s^i]_d = [s^i]_d$
   All parties perform
   Check4ConsistentTuples$(l, (P', t'_a, t'_p, t'_f), (\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3), \{[\![s^j]\!]_{j=1}^{n'}\})$
   If the output is a pair of disputed parties or set of crashed parties denoted as $P_{faultyset}$ then all parties take it as output and halt. Otherwise, continue to the next step.
8. Finding disputed parties : Let M_{i^*} be the i^{*th} row of M, where M is the hyper-invertible matrix used in *TripleShareRandom*, such that $[a^{i^*}]_{n'-1} = M_{i^*}([s^1]_{n'-1}, ..., [s^{n'}]_{n'-1})$.
   All parties compute $[_jq^{i^*}]_d = [x^{i^*}]_d + M_{i^*}([_js^1]_d, ..., [_js^{n'}]_d)$ for $j \in \{1,2,3\}$ and send their shares to P_{king}.
   For a certain j, P_{king} finds a k^* where $k^{*^{th}}$ share of $[_jq^{i^*}]_d$ is not equal to the value received from P_{k^*} in evaluate phase and broadcasts $(k^*, corrupt)$.
   For party $P_h \in \{P_{king}, P_k^*\}$, the parties perform heartbeat protocol. If both the parties P_{king}, P_{k^*} are alive, then all parties in P' consider $E = P_{king}, P_{k^*}$ as output. Otherwise set E as parties for which were detected as crashed by heartbeat protocol.

Lemma 10. ***AdditionMultiplicationGateEval*** *securely computes addition and multiplication gates for a given segment seg in the presence of mixed adversary provided that (P', t'_a, t'_p, t'_f) is valid. Also, the communication complexity of* ***AdditionMultiplicationGateEval*** *is* $O(lnk + (t_a + t_f)(2n^2k + BA(k)))$

Proof. Complexity : In step 2, the communication complexity of *GenerateMultiplicationTriples*$(d, d', n'-1, l, (P', t'_a, t'_p, t'_f))$. is $O(lnk + n^2k + BA(k))$. In step 3, for each multiplication P_{king} receives and sends $O(nk)$ bits.
In step 4, *CheckConsistencyKing* is performed twice where the communication complexity is $O(nk + (t_a + t_f)(n^2k + BA(k)))$.
In step 5, *PubReconRobust* is also performed twice with the communication complexity (public reconstruction complexity n^2k).
In step 6, combined $O(n^2)$ elements are shared to distribute $[_0s^i]_d, [_1s^i]_d, [_2s^i]_d, [_3s^i]_d$.
In step 7, *Check4ConsistentTuples* is invoked once which has complexity of $O(2nk + (t_a + t_f)(2n^2k + BA(k)))$.
In step 8, all parties send $O(n)$ elements to P_{king} and broadcast from P_{king} along with heartbeat takes $O(BA(k))$.
Hence, the communication complexity of c_m multiplication gates is $O(c_mnk + \frac{c_m}{l}(t_a + t_f)(2n^2k + BA(k)))$.

5.4 Output Gate

OutputGateEval uses *ZeroShareRandom* and player-elimination framework. Initially, the parties perform ZeroShareRandom in order to generate l d-sharings of 0. This step is, however, repeated if the output it receives is a pair of disputed parties or a set of crashed parties which are eliminated from the set of active parties. The parties then add their shares of output with the shares of 0 to reconstruct the final output.

Protocol 4: OutputGateEval$(l, (P', t'_a, t'_p, t'_f))$

For each segment of size l :

1. Outputting 0-Random Shares: All parties perform *ZeroShareRandom*. If the output of *ZeroShareRandom* is a pair of disputed parties or set of crashed parties denoted as $P_{faultyset}$ then remove these parties from further computation such that $P' = P' \backslash P_{faulty_set}$ and repeat the above step with modified P'.
   Otherwise continue with l $d-$sharings of 0.
2. Let $P_o \in P'$ be the party which receives the output s. For each output s, $[s]_t$ is $t-$sharing.
   All parties $P \in P'$ compute $[s]_t = [s]_t + [0]_t$ and send $[s]_t$ to P_o.
   Party $P_o \in P'$ now reconstructs the output s from the received shares.

Lemma 11. ***OutputGateEval*** *securely reconstructs the output to the specific party P_o in the presence of mixed adversary provided that (P', t'_a, t'_p, t'_f) is valid. Also, the communication complexity of* ***OutputGateEval*** *is* $O((n + \lceil \frac{c_o}{l} \rceil)(c_o nk + lnk + n^2 k))$

Proof. Complexity : In step 1, *ZeroShareRandom* is performed atmost $O(n) + \lceil \frac{c_o}{l} \rceil$ times. Hence, the overall communication complexity is $O((n + \lceil \frac{c_o}{l} \rceil)(c_o nk + lnk + n^2 k))$

5.5 Main Protocol

In the main protocol, all parties evaluate input gate to share their inputs securely. Further, all parties evaluate random gates to generate random sharings. The circuit is divided into $\lceil c_m/l \rceil$ segments of size l where c_m is number of multiplication gates. Topological order is followed to evaluate segments by invoking *AdditionMultiplicationGateEval* for each segment. If the pair of disputed parties or set of crashed parties is received then these parties are removed from active set of parties P' and the respective segment is evaluated again. At the end, output gates are evaluated to reconstruct output for the required party.

Protocol 5: MainProtocol(C)

Let $P' = P, t'_a = t_a, t'_p = t_p, t'_f = t_f$ and $C = \{c_i, c_r, c_m, c_o\}$.

1. Input Gates : All parties perform *InputGateEval*.
2. Random Gates : All parties perform *RandomGateEval*.
3. Evaluation : All parties agree on partition of circuit into segments $(seg_1, seg_2, ..., seg_{\lceil c_m/l \rceil})$,
   where each segment consists of l multiplication gates independent of multiplication depth. For each segment $j \in \{1, ..., \lceil \frac{c_m}{l} \rceil\}$, perform *AdditionMultiplicationGateEval*$(seg_j, (P', t'_a, t'_p, t'_f))$. If the output is a pair of disputed parties or set of crashed parties denoted as $P_{faultyset}$ then remove these parties from further computation such that $P' = P' \backslash P_{faultyset}$ and repeat the above step with modified P'. Otherwise continue with next value of j.
4. Output Gates : All parties perform *OutputGateEval*$(l, (P', t'_a, t'_p, t'_f))$.

Lemma 12. ***MainProtocol*** *securely realizes any circuit if* $3t_a + 2t_a + t_f < n$ *in the presence of mixed adversary. Also, the communication complexity of* ***MainProtocol*** *is* $O((n + \lceil \frac{c_o}{l} \rceil)(c_o nk + lnk + n^2 k))$

Proof. Correctness and Secrecy : It is implied from the protocols invoked by *Main*. Complexity : For c_i input gates, the communication complexity is $O(c_i(nk + BA(k)))$.
For c_r random gates, the communication complexity is $O((\lceil \frac{c_r}{l} \rceil + (t_a + t_p))(lnk + n^2 k + BA(k)))$.
For c_m multiplication gates, the communication complexity is $O(c_m nk + \lceil \frac{c_m}{l} \rceil (t_a + t_f)(2n^2 k + BA(k)))$.
For c_o output gates, the communication complexity is $O((n + \lceil \frac{c_o}{l} \rceil)(c_o nk + lnk + n^2 k))$.
Hence, the overall communication complexity of evaluation phase is $O((c_i + c_r + c_m + c_o)nk + (c_i + c_r + c_m)(BA(k)) + n^3 k + nBA(k) + {c_o}^2 nk)$.

6 Security

Identical to [15], we establish our requirements for security for mixed adversarial setting. The view of a party is defined as its input values, randomness and the received messages. Let $\mathcal{S}$ be ideal-world adversary and $\mathcal{A}$ be real-world adversary. We define the view of adversary as views of passively and actively corrupted parties. Let $\mathcal{F}$ be ideal-world functionality with domain $(X \cup \{\perp\})^n$.
The ideal world is defined as follows : The correct parties, as defined earlier in adversarial model, takes input value from X. The passively-corrupted parties provide their input to the simulator. Accordingly, simulator chooses the input value for actively-corrupted parites out of $X \cup \{\perp\}$. Also, simulator defines a set D of fail-corrupted parties whose input is set to $\perp$. Now, functionality $\mathcal{F}$ is evaluated using the inputs specified by the simulator.
The security requirement is that for every (unbounded) real-world adversary $\mathcal{A}$, there exists an ideal-world adversary $\mathcal{S}$ such that its joint distribution of its

view and the output of honest parties computed by the ideal functionality $\mathcal{F}$ is equal to the joint distribution of real-world adversary $\mathcal{A}$'s view and the output of honest parties executing the protocol.

7 Conclusion

We have presented an efficient MPC protocol in mixed adversarial setting which is linear in communication complexity and is independent of the multiplicative depth D_m of the circuit over a finite field $\mathcal{F}$. In the previous results, multiplicative depth independent circuit evaluation was achieved only for active adversarial setting[13]. Also, linear communication complexity, dependent on multiplicative depth, was achieved in mixed adversarial setting[15]. An interesting future work would be to study the application of this protocol on various functions which can benefit from the absence of multiplicative depth in the communication complexity.

8 Appendix

8.1 Heartbeat & Player-Elimination Framework

The purpose of this sub-protocol is to adapt player-elimination framework [14] for mixed adversarial setting. Let P_h be a party such that $P_h \in P'$ and all $P_{i \neq h} \in P'$ wants to find out whether P_h is crashed. The sub-protocol **Heartbeat** allows them to reach an agreement on whether P_h is alive.

Protocol 1: Heartbeat$(P_h, (P', t'_a, t'_p, t'_f))$

- The party P_h sends a bit with value 1 to every party $P_{j \neq h} \in P$.
- Every party P_j runs a consensus protocol with an input of value 1 if that's the value it received from P_h otherwise with an input of value 0.
- If the output of the consensus is 1 then, the parties output "alive" otherwise the parties output "crashed".

Protocol 2: PlayerElimination(Π)

1. Initialization : All parties set their happy-bit to "happy"
2. Execution : All parties execute protocol Π
3. Fault detection : Every party send their happy-bit to every other party and sets its own happy-bit to "unhappy" if from at least one party it either receives an "unhappy" bit or does not receive any bit. All parties run consensus on their happy-bits. Every player sets its happy-bit as per the result of the consensus. If the outcome is happy then all parties halt. Otherwise, perform next step.
4. Fault localization : Let $P_r \in P'$ be the party with the smallest index. All parties $P_i \in P_r$ sends to P_r the values generated, sent and received in the above steps. If P_r does not receive values from some parties, it uses some default value. Now, P_r simulates the above steps on the

behalf of each $P_i \in P'$ to check the correctness of the result generated. If P_r identifies a P_i which does not follow the steps, then it broadcasts $(P_i, corrupt)$ and all parties set $E = \{P_r, P_i\}$. If P_i finds a conflict between message index l where P_i should have sent x to P_k but P_k claimed to have received x' such that $x \neq x'$ then P_r broadcasts $(l, P_i, P_k, x, x', disputed)$.
In case of dispute message broadcast by P_r, the conflicting parties P_i and P_k respond by broadcasting their stand (agree/disagree).
If P_i disagrees, all parties set $E = \{P_i, P_r\}$. If P_k disagrees, all parties set $E = \{P_k, P_r\}$. Otherwise, every party sets $E = \{P_i, P_k\}$.

5. Crash-check fault localization : For party $P_h \in \{P_i, P_k, P_r\}$, the parties perform heartbeat protocol.
6. Output : If all the parties P_i, P_r and P_k are alive, then all parties in P' consider E as output. Otherwise set E as parties for which were detected as crashed by heartbeat protocol.

References

1. Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.
2. Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In *Theory of Cryptography Conference*, pages 213–230. Springer, 2008.
3. Eli Ben-Sasson, Serge Fehr, and Rafail Ostrovsky. Near-linear unconditionally-secure multiparty computation with a dishonest minority. In *Annual Cryptology Conference*, pages 663–680. Springer, 2012.
4. Piotr Berman, Juan A Garay, and Kenneth J Perry. Bit optimal distributed consensus. In *Computer science*, pages 313–321. Springer, 1992.
5. 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.
6. Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. *Secure multiparty computation*. Cambridge University Press, 2015.
7. Ivan Damgård and Yuval Ishai. Scalable secure multiparty computation. In *Annual International Cryptology Conference*, pages 501–520. Springer, 2006.
8. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference*, pages 572–590. Springer, 2007.
9. Matthias Fitzi, Martin Hirt, and Ueli Maurer. Trading correctness for privacy in unconditional multi-party computation. In *Annual International Cryptology Conference*, pages 121–136. Springer, 1998.
10. Juan A Garay and Kenneth J Perry. A continuum of failure models for distributed computing. In *International Workshop on Distributed Algorithms*, pages 153–165. Springer, 1992.
11. Daniel Genkin, Yuval Ishai, Manoj M Prabhakaran, Amit Sahai, and Eran Tromer. Circuits resilient to additive attacks with applications to secure computation. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 495–504, 2014.

12. 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.
13. Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional mpc with guaranteed output delivery. In *Annual International Cryptology Conference*, pages 85–114. Springer, 2019.
14. Martin Hirt, Ueli Maurer, and Bartosz Przydatek. Efficient secure multi-party computation. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 143–161. Springer, 2000.
15. Martin Hirt and Marta Mularczyk. Efficient mpc with a mixed adversary. In *1st Conference on Information-Theoretic Cryptography (ITC 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
16. Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In *Annual International Cryptology Conference*, pages 463–482. Springer, 2006.
17. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
18. Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.