# 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
{achintya.desai@research.,shubham.raj@research.,srinathan@}iiit.ac.in

**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.[14] 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.[16] which proposed an efficient MPC protocol tolerating mixed adversary with communication complexity of $O((c_i + c_m + c_o)n\kappa + c_i BA(\kappa) + D_m(n^3\kappa + nBA(\kappa)))$ bits, where $D_m$ is the multiplicative depth of the circuit and $\kappa$ is the size of an element in the field. Additionally, Hirt et al.[16], 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 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)[19]. 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.[10] tried reaching a middle ground and pointed out a trade-off between the adversarial capabilities and the allowed number of corruptions. Fitzi et al.[10] 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 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. [13] and Chaum et al. [5] presented the first protocols for MPC in pure setting and Fitzi et al.[10] 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], [17], [9], [2], [3], [12], [14], [16] making the communication complexity per multiplication gates linear in terms of the number of parties. Hirt et al.[16] studied the efficiency of a secure

| **Reference Papers** | **Adversarial Setting** | **Communication Complexity** | **Dependency on Multiplicative Depth** |
|---|---|---|---|
| Beerliová-Trubíniová et al.[2] | Active | linear | ✓ |
| Goyal et al.[14] | Active | linear | ✗ |
| Hirt et al.[16] | Mixed | linear | ✓ |
| Our Paper | Mixed | linear | ✗ |

**Table 1.** MPC variations

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.[16] 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.

### 1.1 Contributions

This paper improves the work done by Hirt et al.[15] in the mixed adversarial setting. The communication complexity of evaluating all the multiplication gates is $O(c_m * n\kappa + D(n^3\kappa + nBA(1)))$ where $c_m$ is the number of multiplication gates and $D$ is the multiplicative depth of the circuit. Most of the works involving active setting also suffer from this term [9],[8],[3]. As suggested in Hirt et al.[15], we remove the inherent dependency on multiplicative depth $D$ by combining their technique with the technique proposed by Goyal et al.[14]. In this technique, the public reconstruction protocol for intermediate result is replaced with a reconstruction protocol involving a single party $P_{King}$ which reconstructs the intermediate result and broadcasts it to all parties. The work done by Goyal et. al. was applicable to active setting. This paper transforms the technique proposed by Goyal et. al. to work in mixed adversarial setting. We also provide simulation-based proofs for the constructions involved in the technique.

There are two main changes adopted to work under mixed adversarial setting. Firstly, the protocol proposed by Goyal et. al. works for active adversary setting with threshold $t_a$. In this paper, we handle active, passive as well as fail corrupt adversary. Hence, the sub-protocols for player-elimination framework, input gate and public reconstruction are followed from Hirt et. al.[16] upon which the technique proposed by Goyal et. al. is constructed. Secondly, during the reconstruction of intermediate result, $P_{King}$ plays a significant role in reconstruction and distribution of the result. This party $P_{King}$, like others, is also susceptible to crash fault. For handling this case, we perform a heartbeat protocol with $P_{King}$ from all the parties to ensure that the results are correctly reconstructed and distributed among the parties before the verification proceeds. The rest of the changes involving simulation and functionality, ensure that the security of the protocol remains unhindered.

It might seem that the solution is lacking novelty but answering this open question is considered to be an important addition to the existing MPC literature. Our protocol is useful specifically in the cases where circuit is "narrow" in terms of multiplication depth.

### 1.2 Structure

There are mainly four sections including introduction. Section 2 establishes the necessary preliminaries such as byzantine agreement, Heartbeat and 4-consistency etc. Section 3 and 4 presents preparation and evaluation phase respectively.

Each section consists of ideal functionalities. Each ideal functionality is then defined and realized in real world using combination of necessary protocols and hybrid functionalities which can further be replaced with their real world protocols. At the end of each functionality, a simulation-based proof and complexity is provided to prove the security of individual functionality and thereby the entire technique.

## 2 Preliminaries & Basic Setting

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. Since the protocols are executed withing player-elimination framework, some parties are eliminated as the protocol progresses. This set of current parties is denoted as $P'$. We define a configuration as $(P', t'_a, t'_p, t'_f)$ where $t'_a, t'_p, t'_f$ are the respective counterparts of the corruption thresholds and $n' = |P'|$. A configuration $(P', t'_a, t'_p, t'_f)$ is valid (i) if $P' \subseteq P$ due to player-elimination framwork, (ii) if $t_a + t_p < n' - 2t'_a - t'_f$ since we require Shamir secret sharings with degree $d = t_a + t_p$ be even reconstructable by parties in $P'$ and (iii) $|P'_a| \leq t'_a$, $|P'_p| \leq t'_p$, $|P'_f| \leq t'_f$. Also, let $\Lambda$ be the default input when a party does not provide any input.

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. This work also requires understanding of Byzantine agreement[11], Circuit randomization[1] and Hyper Invertible matrix[2] as a prerequisite whose extensive study material can be found in [6].

Complete break down : When the configuration is invalid or the values are not consistent, this protocol is executed. It stops the execution and sends the inputs to adversary, which then can evaluate all outputs. In such cases, no security can be guaranteed since the prerequisite conditions are not met. Hence, simulator construction is not required.

### 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 [11] 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 [11] 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 [11] 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 [11], [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 [11], [4] is $O(n^2)$ where n is the size of the initial set $\mathcal{P}$. Let $BA(\kappa)$ be the complexity of broadcasting or reaching consensus on a $k-bit$ message in mixed setting. Hence using the aforementioned protocols gives us $BA(\kappa) = O(n^2\kappa)$.

### 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 triplet $[a]_d$, $[b]_d$, $[c]_d$ such that $c = ab$ which has been pre-shared is available. In this technique, we first prepare $c_m$ shared multiplication triplets $[a]_d$, $[b]_d$, $[c]_d$ 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 $q = x + a$ and $e = y + b$, then, $z = qe - qb - ae + c$, where $(a, b, c)$ is a multiplication triplet. If the multiplication triplet $(a, b, c)$ is random then $q$ and $e$ are random values irrespective of $x$ and $y$. Hence, a sharing $[z]_d$ can be linearly computed as $[z]_d = qe - q[b]_d - e[a]_d + [c]_d$, by reconstructing $[q]_d = [x]_d + [a]_d$ and $[e]_d = [y]_d + [b]_d$.

### 2.4 Secret Sharing

In this section, we are establishing secret sharing variations, similar to [16], as a fundamental building block for further protocols. We use Shamir's secret sharing scheme[18] 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 $\beta$ 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 $l-1$ 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

[14] 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 $\pi$ 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 $\pi$ 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 $[_jr]_{t_a}$. [14]*

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[14]*

### 2.6 Heartbeat & Player-Elimination Framework

**Protocol 5: 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".

The purpose of this sub-protocol is to adapt player-elimination framework [15] 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 6: PlayerElimination**($\Pi$)

1. Initialization : All parties set their happy-bit to "happy"
2. Execution : All parties execute protocol $\Pi$
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 $(P_i, \mathit{corrupt})$ 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 $(l, P_i, P_k, x, x', \mathit{disputed})$.
   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 $E = \{P_i, P_k\}$.
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.

## 3 Preparation Phase

### 3.1 The functionality $\mathcal{F}_{triplets}$

**Functionality** $\mathcal{F}_{triplets}$

The functionality receives set of corrupted parties $P_a, P_p, P_f$. Let $d = t_a + t_p$.

1. Receive $l$ and a valid configuration $(P', t'_a, t'_p, t'_f)$ from each party.
2. If the received values are not consistent (except $\Lambda$) or the configuration is invalid then execute Complete Break Down. Otherwise send configuration $(P', t'_a, t'_p, t'_f)$, $l$ to the adversary and proceed.
3. Adversary can send following three types of messages:

4. If adversary sends (TRIPLETS, $(([a_j^k]_d, [b_j^k]_d, [c_j^k]_d))_{k=1,...,l,P_j \in (P_a \cup P_p)}$, $(([a_j^k]_{n'-1}, [b_j^k]_{n'-1}))_{k=1,...,l,P_j \in (P_a \cup P_p)}$ then generate $l$ multiplication triplets among parties in $P'$ by simulating the shares of honest parties. For each multiplication triplet $k$, choose random $a^k, b^k$ such that $c^k = a^k.b^k$. For each $k = 1, ..., l$, choose polynomials $g_a^k, g_b^k, g_c^k$ at random from set of polynomials with degree $d$ such that $g_a^k, g_b^k, g_c^k$ goes through $(0, a^k), (0, b^k), (0, c^k)$ respectively and all the points in set $\{(\alpha_j, [a_j^k]_d) | P_j \in (P_a \cup P_p)\}$, $\{(\alpha_j, [b_j^k]_d) | P_j \in (P_a \cup P_p)\}$, $\{(\alpha_j, [c_j^k]_d) | P_j \in (P_a \cup P_p)\}$ respectively. Also, for each $k = 1, ..., l$, choose polynomials $h_a^k, h_b^k$ at random from set of polynomials with degree $n' - 1$ such that $h_a^k, h_b^k$ goes through $(0, a^k), (0, b^k)$ respectively and all the points in set $\{(\alpha_j, [a_j^k]_{n'-1}) | P_j \in (P_a \cup P_p)\}$ and $\{(\alpha_j, [b_j^k]_{n'-1}) | P_j \in (P_a \cup P_p)\}$ respectively. Send to each $P_i \in P'$ its share of triplets $g_a^k(\alpha_i), g_b^k(\alpha_i), g_c^k(\alpha_i)$ and $h_a^k(\alpha_i), h_b^k(\alpha_i)$.
5. If adversary sends (ACTIVESET,$E$), where $E \subseteq P', |E \cap P_a| \geq \frac{|E|}{2}$ then Send (ACTIVESET, $E$) to the parties.
6. If adversary sends (CRASHSET,$E$), where $E \subseteq P', E \subseteq P_f \cup P_a$ and $E \neq \emptyset$ then Send (CRASHSET, $E$) to the parties.
7. Otherwise treat message as (TRIPLETS, $((0, 0, 0))_{k=1,...,l,P_j \in (P_a \cup P_p)}$, $((0, 0))_{k=1,...,l,P_j \in (P_a \cup P_p)}$) and goto step 4.

### 3.2 Realizing functionality $\mathcal{F}_{triplets}$

**Generating Random Triplet-Sharings** Identical to [14], this protocol *TripletShareRandom* generates $l$ random triplet 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: TripletShareRandom**$(d, d', d'', l, (P', t'_a, t'_p, t'_f))$

Generate each bucket of size $\mathcal{L} = n' - 2t'_a - t'_p - min(t'_a, t'_p)$ using following protocol:

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 triplet 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 $\mathcal{L}$ sharings as output of the bucket i.e. $[r^{n'-\mathcal{L}+1}]_{d,d',d''}, ..., [r^{n'}]_{d,d',d''}$.

**Lemma 2.** ***TripletShareRandom*** *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* ***TripletShareRandom*** *is $O(ln'\kappa + n'^2\kappa)$*

*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 $\mathcal{L} \leq n' - 2t'_a - t'_p - \min(t'_a, t'_p)$ values of $r^i$ and $\mathcal{L}$ 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 $O(n'^2\kappa)$ bits of communication($n'$ shares are broadcasted to $n'$ parties). Each bucket execution generates $\mathcal{L} \leq n' - 2t'_a - t'_p - \min(t'_a, t'_p)$ triple-sharings. Since $3t'_a < n'$ and $2t'_p < n'$, $\mathcal{L} \leq n' - 2t'_a - t'_p - \min(t'_a, t'_p)$ at least $\frac{1}{5}n'$. Therefore, the complexity of TripletShareRandom is $O((\frac{5l}{n'} + 1)n'^2\kappa) = O(ln'\kappa + n'^2\kappa)$

**Generating Random Multiplication Tuples** Goyal et al. [14] described the steps to generate multiplication tuples, which we have inherited in this

work. Initially, all the parties in $P'$ execute TripletShareRandom 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 $n'-1$ except $c$ for which $n'-1$ shares are discarded.

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

1. Generate three random triplets : All parties in $P_i \in P'$ invoke *TripletShareRandom*$(d, d', n'-1, l, (P', t'_a, t'_p, t'_f))$, *TripletShareRandom*$(d, d', n'-1, l, (P', t'_a, t'_p, t'_f))$ and *TripletShareRandom*$(d, 2d', n'-1, l, (P', t'_a, t'_p, t'_f))$, where $d' = t'_a + t'_p$, in parallel to generate random triplet 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 *PubRecon* 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 $([a^1]_{d,n'-1}, [b^1]_{d,n'-1}, [c^1]_d), ..., ([a^l]_{d,n'-1}, [b^l]_{d,n'-1}, [c^l]_d)$

**Lemma 3.** ***GenerateMultiplicationtriplets*** *detectably generates $l$ correct triplets 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* ***GenerateMultiplicationtriplets*** *is $O(ln'\kappa + n'^2\kappa)$*

*Proof.* Correctness and Secrecy : Correctness and secrecy follows from lemma *TripletShareRandomlemma* and *PubRecon* 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 *TripletShareRandom* and *PubRecon*, the complexity $O(ln'\kappa + n'^2\kappa)$ follows from lemma 3.2 and *PubRecon*.

### Realising $\mathcal{F}_{triplets}$

**Theorem 1.** *Assuming that $3t_a + 2t_a + t_f < n$, the protocol PlayerElimination(GenerateMultiplicationtriplets) securly evaluates $\mathcal{F}_{triplets}$ in the presence of a static mixed adversary.*

*Proof.* We construct a simulator $S_{triplets}$ towards real-world adversary $\mathcal{A}$. It interacts with ideal-world functionality with a black-box access to $\mathcal{A}$ and executing the protocol. We have to prove that output is distributed correctly in both(real and ideal) worlds.

**Simulator $\mathcal{S}_{triplets}$**

1. Receive $(P', t'_a, t'_p, t'_f)$, $l$ from $\mathcal{F}_{triplets}$.
2. Execute PlayerElimination(GenerateMultiplicationtriplets$((d, l, (P', t'_a, t'_p, t'_f))))$ till Fault Detection step on behalf of correct parties in $P'$ excluding actively corrupted parties.
   In each round, send the messages generated by the protocol to $\mathcal{A}$ and receive the messages from corrupted parties & keep track of crashed parties.
3. If the parties do not agree on the output of Fault Detection phase then abort.
4. Otherwise if the output of fault detection step was happy then For each triplet $k = 1, ..., l$, compute the shares of the corrupted parties in $P'$ that would result from the protocol depending on messages exchanged during protocol execution. Choose random shares for the corrupted parties not in $P'$. Send these shares combined as (TRIPLETS, shares) to $\mathcal{F}_{triplets}$.
5. If the output of fault detection step was unhappy then continue executing the next step (fault localization). If the honest parties do not agree on the resulting set $E$ then abort. Otherwise, send (ACTIVESET, $E$) or (CRASHSET,$E$) to $\mathcal{F}_{triplets}$ depending on the outcome of fault localization step.

If the honest parties do not input consistent values or valid configuration then the simulator aborts and Complete Break Down is executed. Assume that the honest parties input consistent values $l, P', t'_a, t'_p, t'_f$ and valid set $P'$. Due to consensus nature of fault detection, simulator does not abort at step 3. Simulator sends tripletS command to the functionality $\mathcal{F}_{triplets}$ if and only if the result of fault detection step is happy which is also same in real world implying that no fault caught in this computation.

Assume that the simulator sends tripletS command which implies, in an real world, that no party crashed and no honest party is unhappy after executing GenerateMultiplicationtriplets. From lemma 3, shared values $a$ and $b$ are uniformly random and independent of the view of adversary. It also guarantees the correctness of $c = ab$. Hence, the triplets are distributed identically compared to the ideal world. It can also be seen that in both worlds, real and ideal, the polynomials are uniformly random and consistent with the shares outputted by corrupted parties and the triplet value(a,b,c). Otherwise, at least one honest party gets unhappy.

Since there are no private inputs and deterministic nature of ideal functionality, correctness need to be proven. For correctness, it is only required to prove that the set $E$ is generated correctly compared to ideal world. In case of CRASHSET, the completeness of Heartbeat protocol 2.6 ensures that the set $E$ contains no correct party. In case of ACTIVESET, the set $E$ always consists of pair of parties making conflicting claims from $\{P_i, P_j\}, \{P_r, P_j\}, \{P_i, P_r\}$.

### 3.3 The functionality $\mathcal{F}_{Preparation}$

This functionality takes $l$ number of triplets and current configuration $(P', t'_a, t'_p, t'_f)$ as input. Adversary sends a valid set $P'' \subseteq P'$ with thresholds $t''_a, t''_p, t''_f$ which could be smaller subset of parties that do not include parties eliminated by player elimination framework for being fail-corrupt. The adversary also sends the shares of actively and passively corrupted parties. The new configuration $(P'', t''_a, t''_p, t''_f)$ is sent to all parties with their share of $l$ multiplication triplets of degree $d = t_a + t_p$ and $n' - 1$.

**Functionality** $\mathcal{F}_{Preparation}$

The functionality receives set of corrupted parties $P_a, P_p, P_f$. Let $d = t_a + t_p$

1. Receive $l$ and a valid configuration $(P', t'_a, t'_p, t'_f)$ from each party.
2. If the received values are not consistent (except $\Lambda$) or the configuration is invalid then execute Complete Break Down. Otherwise send configuration $(P', t'_a, t'_p, t'_f)$, $l$ to the adversary and proceed.
3. Adversary sends a valid set $P'' \subseteq P'$ with thresholds $t''_a, t''_p, t''_f$ and for each party $P_j \in (P_a \cup P_p)$ sends shares held of each triplet i.e. $(([a^k_j]_d, [b^k_j]_d, [c^k_j]_d))_{k=1,...,l}$ and $(([a^k_j]_{n'-1}, [b^k_j]_{n'-1}))_{k=1,...,l}$. If values are not received, assume $P'' = P', t''_a = t'_a, t''_p = t'_p, t''_f = t'_f, (([a^k_j]_d, [b^k_j]_d, [c^k_j]_d))_{k=1,...,l} = (0, 0, 0)$, $(([a^k_j]_{n'-1}, [b^k_j]_{n'-1}))_{k=1,...,l} = (0, 0)$.
4. Send the new configuration $(P'', t''_a, t''_p, t''_f)$to all parties.
5. Generate $l$ multiplication triplets among parties in $P''$ by simulting the shares of honest parties. For each multiplication triplet $k$, choose random $a^k, b^k$ such that $c^k = a^k.b^k$. For each $k = 1, ..., l$, choose polynomials $g^k_a, g^k_b, g^k_c$ at random from set of polynomials with degree $d$ such that $g^k_a, g^k_b, g^k_c$ goes through $(0, a^k), (0, b^k), (0, c^k)$ respectively and all the points in set $\{(\alpha_j, [a^k_j]_d) | P_j \in (P_a \cup P_p)\}$, $\{(\alpha_j, [b^k_j]_d) | P_j \in (P_a \cup P_p)\}$, $\{(\alpha_j, [c^k_j]_d) | P_j \in (P_a \cup P_p)\}$ respectively. Also, for each $k = 1, ..., l$, choose polynomials $h^k_a, h^k_b$ at random from set of polynomials with degree $n' - 1$ such that $h^k_a, h^k_b$ goes through $(0, a^k), (0, b^k)$ respectively and all the points in set $\{(\alpha_j, [a^k_j]_{n'-1}) | P_j \in (P_a \cup P_p)\}$ and $\{(\alpha_j, [b^k_j]_{n'-1}) | P_j \in (P_a \cup P_p)\}$ respectively.
6. Send to each $P_i \in P''$ its share of triplets $g^k_a(\alpha_i), g^k_b(\alpha_i), g^k_c(\alpha_i)$ and $h^k_a(\alpha_i), h^k_b(\alpha_i)$ .

This functionality generates $l$ number of triplets and handles case of failure by identifying set containing actively corrupted or fail corrupted parties. The parties input $l$ number of triplets and current configuration $(P', t'_a, t'_p, t'_f)$. Based on adversary action, one of the following outcomes occur:
(i)The computation succeeds, if adversary provides its own shares, by computing and sharing $l$ multiplication triplets with degree $d = t_a + t_p$ and $n' - 1$ among $P'$ parties.
(ii) The actively corrupted parties disrupt the protocol and a set containing an actively corrupt party is identified and given to all parties.
(iii) The actively corrupted parties disrupt the protocol as well as a set of fail-corrupt parties are given to all parties.
If none of the above criteria is satisfied, the parties receive shared triplets with corrupt party shares set to 0.

### 3.4 Realizing functionality $\mathcal{F}_{Preparation}$ in the $\mathcal{F}_{triplets}-$ hybrid model

This protocol realizes $\mathcal{F}_{Preparation}$ in the $\mathcal{F}_{triplets}-$ hybrid model. It divides the number of triplets $L$ into $t_a + t_f$ segments of size $l$ and calls $\mathcal{F}_{triplets}$ sequentially with $l$ as input. If it outputs set $E$ of adversaries then it eliminates parties from $P''$. It results with all parties giving output set $P''$ and parties in $P''$ share of the triplets.

**Protocol 3: PreparationPhase**$(L, (P', t'_a, t'_p, t'_f))$

Let $l = \lceil \frac{L}{t_a + t_f} \rceil, d = t_a + t_p$ Set $P'' = P', t''_a = t'_a, t''_p = t'_p, t''_f = t'_f$. For each segment $k = 1...(t_a + t_f)$ do:

1. Send configuration $(P'', t''_a, t''_p, t''_f)$, $l$ to the functionality $\mathcal{F}_{triplets}$.
2. If $\mathcal{F}_{triplets}$ outputs (ACTIVESET,$E$) then set $P'' = P''$ $E$ and $t''_a = t''_a - |E|/2$ and repeat step 1.
3. If $\mathcal{F}_{triplets}$ outputs (CRASHSET,$E$) then set $P'' = P''$ $E$ and $t''_f = t''_f - |E|$ and repeat step 1.
4. Otherwise save the sharings received from $\mathcal{F}_{triplets}$ and repeat step 1 with next segment $k + 1$. If all segments are evaluated then continue.
5. Every party outputs new configuration $(P'', t''_a, t''_p, t''_f)$ and remaining parties in $P''$ output last $l$ triplets of shares received from $\mathcal{F}_{triplets}$.

**Theorem 2.** *Assuming that $3t_a + 2t_p + t_f < n$, the above protocol securely realizes $\mathcal{F}_{Preparation}$ in the $\mathcal{F}_{triplets}-$hybrid model, in the presence of mixed adversary.*

*Proof.* To prove the security of the above protocol, we need a simultor which will interacts with adversary in ideal world. Functionality $\mathcal{F}_{triplets}$ performs this simulation. Further, we need to prove that the output in real world and ideal world are same. In the real world execution, following invariants are

followed : (i)$(P'', t''_a, t''_p, t''_f)$ is a valid configuration, (ii)The shares outputted by parties in $P''$ form correct $d-$ sharing and $(n'-1)-$ sharings of triplets. Configuration is updated only when parties are eliminated. Since whenever parties are eliminated, $n'' - 2t''_a - t''_f$ is preserved by the protocol implying $3t''_a + 2t''_p + t''_f < n''$. Since $d$ is constant and $(n'-1)$ is set at the beginning of the execution($n''$ changes), a $d$ sharing in $P'$ and $n'-1$ sharing in $P'$ remain correct in $P''$.

## 4 Evaluation Phase

### 4.1 Functionality $\mathcal{F}_{4-Consistency}$

**Functionality $\mathcal{F}_{4-Consistency}$**

The functionality receives set of corrupted parties $P_a, P_p, P_f$. Let $d = t_a + t_p$ and $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ be partitions of $P'$.

1. Receive a valid configuration $(P', t'_a, t'_p, t'_f)$, $l$(number of random tuples) from each party and $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ such that $\mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 = P'$ and $|\mathcal{P}_1|, |\mathcal{P}_2|, |\mathcal{P}_3| \leq d+1$.
2. If the received values are not consistent (except $\Lambda$) or the input configuration is invalid then execute Complete Break Down. Also, if $\mathcal{P}_1 \cup \mathcal{P}_2 \cup \mathcal{P}_3 \neq P'$ or $|\mathcal{P}_1|, |\mathcal{P}_2|, |\mathcal{P}_3| > d+1$ then execute Complete Break Down. Otherwise send configuration $(P', t'_a, t'_p, t'_f)$, $l$ and $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ to the adversary and proceed.
3. If adversary sends (TUPLES, $([_0r^k_j]_d, [_1r^k_j]_d, [_2r^k_j]_d, [_3r^k_j]_d)_{k=1,...,l, P_j \in (P_a \cup P_p)}$), then generate $l$ tuples among parties in $P'$ by simulating the shares of honest parties. For every $k \in 1, ..., l$, choose polynomials $g^k_0, g^k_1, g^k_2, g^k_3$ at random from set of polynomials with degree $d$ such that $g^k_0, g^k_1, g^k_2, g^k_3$ goes through all the points in set $\{(\alpha_j, [_0r^k_j]_d | P_j \in (P_a \cup P_p)\}$, $\{(\alpha_j, [_0r^k_j]_d | P_j \in (P_a \cup P_p)\}$, $\{(\alpha_j, [_0r^k_j]_d | P_j \in (P_a \cup P_p)\}$ and $\{(\alpha_j, [_0r^k_j]_d | P_j \in (P_a \cup P_p)\}$ respectively. Send to every player $P_i \in P'$, the $i$th share of $[\![r^k]\!] = ([_0r^k_i]_d, [_1r^k_i]_d, [_2r^k_i]_d, [_3r^k_i]_d)$ for $k \in 1, ..., l$.
4. If adversary sends (ACTIVESET,$E$), where $E \subseteq P', |E \cap P_a| \geq \frac{|E|}{2}$ then Send (ACTIVESET, $E$) to the parties.
5. If adversary sends (CRASHSET,$E$), where $E \subseteq P', E \subseteq P_f \cup P_a$ and $E \neq \emptyset$ then Send (CRASHSET, $E$) to the parties.
6. Otherwise treat the message as (TUPLES, $(0,0,0,0)_{k=1,...,l, P_j \in (P_a \cup P_p)}$) and goto step 3.

### 4.2 Realizing functionality $\mathcal{F}_{4-Consistency}$

This protocol is obtained from [14] 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 [14]. 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 $P_j$ 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 $O(ln'\kappa + n'^2\kappa)$*

*Proof.* Correctness and Secrecy : Since 4 sharings are distributed instead of 3, the correctness and secrecy follows from lemma 3.2. Complexity : Since the protocol is similar to $TripletShareRandom$, the complexity $O(ln'\kappa + n'^2\kappa)$ follows from lemma 3.2.

**Theorem 3.** *Assuming that $3t_a + 2t_a + t_f < n$, the protocol PlayerElimination(QuadrupleShareRandom) securly evaluates $\mathcal{F}_{4-Consistency}$ in the presence of a static mixed adversary.*

**Simulator** $\mathcal{S}_{4-Consistency}$

*Proof.* 1. Receive $(P', t'_a, t'_p, t'_f)$, $l$ and $\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ from $\mathcal{F}_{4-Consistency}$.
2. Execute PlayerElimination(QuadrupleShareRandom($(l, (P', t'_a, t'_p, t'_f)$,

$(\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3))))$ till Fault Detection step on behalf of correct parties in $P'$ excluding actively corrupted parties.
In each round, send the messages generated by the protocol to $\mathcal{A}$ and receive the messages from corrupted parties & keep track of crashed parties.

3. If the parties do not agree on the output of Fault Detection phase then abort.
4. Otherwise if the output of fault detection step was happy then
For each triplet $k = 1, ..., l$, compute the shares $([_0r_j^k]_d, [_1r_j^k]_d, [_2r_j^k]_d, [_3r_j^k]_d)$ of the corrupted parties in $P'$ that would result from the protocol depending on messages exchanged during protocol execution. Choose random shares for the corrupted parties not in $P'$. Send these shares combined as (TUPLES, shares) to $\mathcal{F}_{4-Consistency}$.
5. If the output of fault detection step was unhappy then continue executing the next step (fault localization). If the honest parties do not agree on the resulting set $E$ then abort. Otherwise, send (ACTIVESET, $E$) or (CRASHSET,$E$) to $\mathcal{F}_{4-Consistency}$ depending on the outcome of fault localization step.

We constructed a simulator $S_{4-Consistency}$ towards real-world adversary $\mathcal{A}$. It interacts with ideal-world functionality with a black-box access to $\mathcal{A}$ and executing the protocol. We have to prove that output is distributed correctly in both(real and ideal) worlds.

If the honest parties do not input consistent values or valid configuration then the simulator aborts and Complete Break Down is executed. Assume that the honest parties input consistent values $l, P', t'_a, t'_p, t'_f$ and valid set $P'$. Due to consensus nature of fault detection, simulator does not abort at step 3. Simulator sends TUPLES command to the functionality $\mathcal{F}_{4-Consistency}$ if and only if the result of fault detection step is happy which is also same in real world implying that no fault caught in this computation.

Assume that the simulator sends TUPLES command which implies, in an real world, that no party crashed and no honest party is unhappy after executing QuadrupleShareRandom. From lemma 4, shared value $r$ is uniformly random and independent of the view of adversary. Hence, the tuples are distributed identically compared to the ideal world. It can also be seen that in both worlds, real and ideal, the polynomials are uniformly random and consistent with the shares outputted by corrupted parties. Otherwise, at least one honest party gets unhappy.

Since there are no private inputs and deterministic nature of ideal functionality, only correctness need to be proven. For correctness, it is only required to prove that the set $E$ is generated correctly compared to ideal world. In case of CRASHSET, the completeness of Heartbeat protocol 2.6 ensures that the set $E$ contains no correct party. In case of ACTIVESET, the set $E$ always consists of pair of parties making conflicting claims from $\{P_i, P_j\}, \{P_r, P_j\}, \{P_i, P_r\}$.

### 4.3 Functionality $\mathcal{F}_{Multiplication}$

**Functionality $\mathcal{F}_{Multiplication}$**

The functionality receives set of corrupted parties $P_a, P_p, P_f$. Let $d = t_a + t_p$

1. Receive $l$(number of multiplications in a segment) and a valid configuration $(P', t'_a, t'_p, t'_f)$ from each party. From each party $P_i \in P'$ receive $d$ & $(n'-1)$ shares of $(x^k, y^k, a^k, b^k, c^k)_{k=1...l}$ where $x^k, y^k$ are inputs to the gate and $(a^k, b^k, c^k)$ are the multiplication triplet.
2. If the received values are not consistent (except $\Lambda$) or the input configuration is invalid then execute Complete Break Down. Also, if the received shares of $x^k, y^k, a^k, b^k, c^k$ do not make a correct $d$-sharings or triplets for any value of $k$ then execute Complete Break Down.
3. For each $k = \{1, ..., l\}$, choose polynomials $g^k_a, g^k_b, g^k_c, g^k_x, g^k_y$ at random from set of polynomials with degree $d$ such that $g^k_a, g^k_b, g^k_c, g^k_x, g^k_y$ goes through $(0, a^k), (0, b^k), (0, c^k), (0, x^k), (0, y^k)$ respectively and all the points in set $\{(\alpha_j, [a^k_j]_d) | P_j \in P'\}$, $\{(\alpha_j, [b^k_j]_d) | P_j \in P'\}$, $\{(\alpha_j, [c^k_j]_d) | P_j \in P'\}$, $\{(\alpha_j, [x^k_j]_d) | P_j \in P'\}$, $\{(\alpha_j, [y^k_j]_d) | P_j \in P'\}$ respectively. Also, for each $k = 1, ..., l$, choose polynomials $h^k_a, h^k_b$ at random from set of polynomials with degree $n'-1$ such that $h^k_a, h^k_b$ goes through $(0, a^k), (0, b^k)$ respectively and all the points in set $\{(\alpha_j, [a^k_j]_{n'-1}) | P_j \in P'\}$ and $\{(\alpha_j, [b^k_j]_{n'-1}) | P_j \in P'\}$ respectively.
4. Send to adversary $l$, $(P', t'_a, t'_p, t'_f)$. Compute & send, for all $k = \{1, ..., l\}$, $h^k_q = g^k_x + h^k_a$ and $h^k_e = g^k_y + h^k_b$ to the adversary. Also, send, for all $k = \{1, ..., l\}$, $(g^k_a(\alpha_j), g^k_b(\alpha_j), g^k_c(\alpha_j), h^k_a(\alpha_j), h^k_b(\alpha_j))_{P_j \in (P_a \cup P_p)}$.
5. Receive a valid configuration $(P'', t''_a, t''_p, t''_f)$ such that $P'' \subseteq P'$ and for each player $k = 1, ..., l$, shares of $(z^k_j)_{P_j \in P_a \cup P_p}$. If no such values are received then set $(P'' = P', t''_a = t'_a, t''_p = t'_p, t''_f = t'_f), (z^k_j) = 0$.
6. Send the new configuration $(P'', t''_a, t''_p, t''_f)$to all parties and for each $k = 1, ..., l$ choose a polynomial $g^k_z$ at random from set of polynomials with degree $d$ such that $g^k_z$ goes through $(0, g^k_x(0).g^k_y(0))$ respectively and all the points in set $\{(\alpha_j, [z^k_j]_d) | P_j \in (P_a \cup P_p)\}$. Send the share $g^k_z(\alpha_i)$ to each $P_i \in P''$. For next steps set $P' = P'', t'_a = t''_a, t'_p = t''_p, t'_f = t''_f$.
7. Now, adversary sends following three types of messages:
8. If adversary sends $(([q^k_j]_0, [e^k_j]_0)_{k=1,...,l, P_j \in (P_a \cup P_p)}$ where $q^k = h^k_q(0)$ and $e^k = h^k_e(0)$, then the send the shares for $k = 1, ..., l$, $([q^k_i]_0, [e^k_i]_0)$ to all parties $P_i \in P''$ such that $[q^k_j]_0 = [q^k_i]_0$ and $[e^k_j]_0 = [e^k_i]_0$.
9. If adversary sends (ACTIVESET,$E$), where $E \subseteq P', |E \cap P_a| \geq \frac{|E|}{2}$ then Send (ACTIVESET, $E$) to the parties.
10. If adversary sends (CRASHSET,$E$), where $E \subseteq P', E \subseteq P_f \cup P_a$ and $E \neq \emptyset$ then Send (CRASHSET, $E$) to the parties.

11. Otherwise treat message as $((0,0))_{k=1,...,l,P_j \in (P_a \cup P_p)}$, and goto step 7.
12. For $k = \{1, ..., l\}$, compute $g_q^k = g_x^k + g_a^k$ and $g_e^k = g_y^k + g_b^k$
    Receive from adversary new configuration $(P'', t_a'', t_p'', t_f'')$ such that $P'' \subseteq P'$.
    If, for all $k = \{1, ..., l\}$, $h_q^k(0) = g_q^k(0)$ and $h_e^k(0) = g_e^k(0)$ then consider $g_z^k(\alpha_i)$ sent in step 6 to each $P_i$ as output. Otherwise, continue with next steps.
    Set $(P' = P'', t_a' = t_a'', t_p' = t_p'', t_f' = t_f'')$
13. This step is explained in the simulator and functionality receives from adversary $\mathcal{A}$, $[_j s^i]_d$ for $j \in \{1, 2, 3\}$ and $P_i \in \mathcal{H} \setminus \mathcal{H}'$ where parties in $\mathcal{H}'$ have been checked by non-corrupt parties only.
    Each party $P_i \in P'$, choose polynomials $g_{_1 s^i}, g_{_2 s^i}, g_{_3 s^i}$ at random from set of polynomials with degree $d$ such that for $j \in \{1, 2, 3\}$, $g_{_j s^i}$ goes through $\{(\alpha_k, [s_k^i]_{n'-1}) | P_k \in \mathcal{P}_j\}$ and goes through the corrupted shares received.
    Each party $P_i \in P'$ sends to each party $P_k \in P' \setminus \mathcal{P}_j$, $g_{_j s^i}(\alpha_k)$.
14. If adversary sends set $E$ or $P_{faultyset}$ then set $E$ is sent to all parties and repeat the execution of $\mathcal{F}_{Multiplication}$ with updated configuration $P'' = P' \setminus E$,, if received ACTIVESET then
    $t_a'' = t_a' - |E|/2, t_f'' = t_f', t_p'' = t_p'$ else received CRASHSET then
    $t_f'' = t_f' - |E|, t_a'' = t_a', t_p'' = t_p'$. Otherwise, adversary sends all shares of corrupt parties of all random 4-consistent tuples as
    $([_0 r^j]_n, [_1 r^j]_n, [_2 r^j]_n, [_3 r^j]_n)$ for corrupt party $P_n$.
    To send all shares of a 4-consistent tuple to a corrupt party, honestly generate random 4-consistent tuple $[\![u^j]\!]$. For each honest party $P_j$, generate a random 4-consistent tuple such that for every corrupt party $P_n$, $([_0 u^j]_n, [_1 u^j]_n, [_2 u^j]_n, [_3 u^j]_n) =$
    $([_0 s^j]_n, [_1 s^j]_n, [_2 s^j]_n, [_3 s^j]_n) + ([_0 r^j]_n, [_1 r^j]_n, [_2 r^j]_n, [_3 r^j]_n)$.
    Send to adversary all shares of a 4-consistent tuple i.e. for all corrupt $P_n$, $([_0 u^j]_n, [_1 u^j]_n, [_2 u^j]_n, [_3 u^j]_n)$.
    If adversary again sends set $E$ or $P_{faultyset}$ then set $E$ is sent to all parties and repeat the execution of $\mathcal{F}_{Multiplication}$ with updated configuration $P'' = P' \setminus E$,, if received ACTIVESET then
    $t_a'' = t_a' - |E|/2, t_f'' = t_f', t_p'' = t_p'$ else received CRASHSET then
    $t_f'' = t_f' - |E|, t_a'' = t_a', t_p'' = t_p'$. Otherwise, adversary sends OK to continue to the next step.
15. Generate $g_{_j q^{i*}}$ randomly from a set of polynomials with degree $d$ such that $g_{_j q^{i*}}(\alpha_k) = [q^{i^*}]_{n'-1}$ for honest party $P_k \in \mathcal{P}_j$. Receive corrupt party shares from $\mathcal{A}$ and forward both i.e. honest and corrupt party shares to $P_{King}$. If $P_{King}$ is honest then it finds a $k^*$ value which does not match with the received shares and equivalent party $P_{k^*}$ is identified and removed from $P'$. If $P_{King}$ is corrupt then receive from $\mathcal{A}$, a set of parties $(P_{King}, P_k^*)$. If parties are alive then

$t''_a = t'_a - |E|/2, t''_f = t'_f, t''_p = t'_p$. Otherwise, whichever party is crashed is removed. Repeat the functionality with updated configuration.

### 4.4 Realizing functionality $\mathcal{F}_{Multiplication}$

**Checking Consistency** Like [14] version of *CheckConsistencyKing*, it is used to check whether $P_{King}$ is corrupt or not during the execution of *MultiplicationGateEval*. 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 2: 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 5.** ***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 $O(n\kappa + (t_a + t_f)(n^2\kappa + BA(\kappa)))$*

*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 $O(n\kappa + (t_a + t_f)(n^2\kappa + BA(\kappa)))$. .

**Checking 4-Consistent Tuples**

**Protocol 3: 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 : Send a configuration $(P', t'_a, t'_p, t'_f)$, $l$ and $(\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3)$ from each party to functionality $\mathcal{F}_{4-Consistency}$.
2. If $\mathcal{F}_{4-Consistency}$ outputs (ACTIVESET,$E$) then set $P'' = P'' \setminus E$ and $t''_a = t''_a - |E|/2$ and halt.
3. If $\mathcal{F}_{4-Consistency}$ outputs (CRASHSET,$E$) then set $P'' = P'' \setminus E$ and $t''_f = t''_f - |E|$ and halt.
4. Otherwise every party saves the shares of 4-consistent tuples received from $\mathcal{F}_{4-Consistency}$ denoted as $\{[\![r^1]\!], ..., [\![r^l]\!]\}$.
5. Distributing Shares : All parties send the shares of $[\![r^j]\!]$ to $P_j$ and compute $[\![u^j]\!] = [\![s^j]\!] + [\![r^j]\!]$ where $j \in \{1, ..., n'\}$
6. 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 $j \in \{0, ..., \lceil \frac{n'}{l} \rceil - 1\}$.
7. 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.

8. 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 $(P_i, corrupt)$. 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 $E = \{P_i, P_k\}$.
9. Crash-check fault localization : For party $P_h \in \{P_i, P_k, P_r\}$, the parties perform heartbeat protocol.
10. 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.

*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.

**Lemma 6.** ***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* $O(2n\kappa + (t_a + t_f)(2n^2\kappa + BA(\kappa)))$

*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
$O(n^2\kappa + (t_a + t_f)(2n^2\kappa + BA(\kappa)))$.

**Realizing $\mathcal{F}_{Multiplication}$ in the $\mathcal{F}_{4-Consistency}$− Hybrid Model** A circuit is divided into multiples segments. For each segment *seg*, *MultiplicationGateEval* computes multiplication gates present. Unlike [16], 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 triplet 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 triplets 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 4: MultiplicationGateEval**

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

1. Receive $l$(number of multiplications in a segment) and a valid configuration $(P', t'_a, t'_p, t'_f)$ from each party. From each party $P_i \in P'$ receive $d$ & $(n' - 1)$ shares of $(x^k, y^k, a^k, b^k, c^k)_{k=1...l}$ where $x^k, y^k$ are inputs to the gate and $(a^k, b^k, c^k)$ are the multiplication triplet.
2. Evaluate gates : For each multiplication gate, we denote input sharings as $[x]_d, [y]_d$ and corresponding multiplication triplet 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 $[z]_d = qe - q[b]_d - e[a]_d + [c]_d$.
   After computing all $l$ multiplication gates, $P_{king}$ performs heartbeat protocol.
   If the output of heartbeat protocol is “alive” then all parties output “Success” and continue. Otherwise parties output “fail” and take

$P_{faultyset} = \{P_{King}\}$ and update the configuration $P' = P' - P_{King}, t'_a = t'_a, t'_a = t'_a, t'_f = t'_f - 1$ and halt.

3. 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.
4. Rechecking the reconstructions : For each multiplication gate, 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 $e^1, ..., e^l$.
   Each party sends $l$,$(P', t'_a, t'_p, t'_f)$ and $[q^1], ..., [q^l]$ to PubReconRobust.
   Every party receives $(P'', t''_a, t''_p, t''_f)$ from PubReconRobust and sets $P' = P'', t'_a = t''_a, t'_p = t''_p, t'_f = t''_f$.
   Each party sends $l$,$(P', t'_a, t'_p, t'_f)$ and $[e^1], ..., [e^l]$ to PubReconRobust.
   Every party receives $(P'', t''_a, t''_p, t''_f)$ from PubReconRobust and sets $P' = P'', t'_a = t''_a, t'_p = t''_p, t'_f = t''_f$.
   Every party now checks whether the reconstructed values $q^1, ..., q^l$ and $e^1, ..., e^l$ from PubReconRobust match with the values distributed by $P_{king}$ in step 2. If all values match then the output shares generated in step 2 are considered correct and shares of output wire are taken as output. Otherwise, continue to the next step with the first incorrect value be denoted as $q^{i^*}$.
5. Rechecking the randomness from multiplication triplets 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 *TripletShareRandom* during evaluation of multiplication triplets.
   Party $P_i$ randomly generates $[_1s^i]_d, [_2s^i]_d, [_3s^i]_d$ such that $k^{th}$ share of $[_js^i]$ is $[s^i_k]_{n'-1}$ 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 $P_k$.
6. 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*$(l, (P', t'_a, t'_p, t'_f), (\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3), \{[\![s^j]\!]_{j=1}^{n'}\})$ which interacts with $\mathcal{F}_{4-Consistency}$.
   If $\mathcal{F}_{4-Consistency}$ outputs (ACTIVESET, $E$) or $\mathcal{F}_{triplets}$ outputs (CRASH-SET, $E$) then all parties take $E$ as output and halt. Otherwise, continue to the next step.

7. Finding disputed parties : Let $M_{i^*}$ be the $i^{*th}$ row of $M$, where $M$ is the hyper-invertible matrix used in *TripletShareRandom*, 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.

**Theorem 4.** *Assuming that $3t_a + 2t_a + t_f < n$, the protocol MultiplicationGateEval securly evaluates $\mathcal{F}_{Multiplication}$ in the $\mathcal{F}_{4-Consistency}$−hybrid model, in the presence of a static mixed adversary. Also, the communication complexity of* ***MultiplicationGateEval*** *is $O(ln\kappa + (t_a + t_f)(2n^2\kappa + BA(\kappa)))$*

*Proof.* We construct a simulator $S_{Multiplication}$ towards real-world adversary $\mathcal{A}$. It interacts with ideal-world functionality with a black-box access to $\mathcal{A}$ and executing the protocol. We have to prove that output is distributed correctly in both worlds.

**Simulator $\mathcal{S}_{Multiplication}$**

Simulator $\mathcal{S}_{Multiplication}$ outputs whatever $\mathcal{A}$ outputs. Emulating $\mathcal{F}_{Multiplication}$

1. Receive $l$, $(P', t'_a, t'_p, t'_f)$, for all $k = \{1, ..., l\}$, $h_q^k$, $h_e^k$, $(g_a^k(\alpha_j), g_b^k(\alpha_j), g_c^k(\alpha_j), h_a^k(\alpha_j), h_b^k(\alpha_j))_{P_j \in (P_a \cup P_p)}$ from $\mathcal{F}_{Multiplication}$. Now, emulating 4.4 as follows
2. For every honest party, simulator send elements such that it lies on the polynomials $h_q^k$ and $h_e^k$ to $P_{king}$. If $P_{king}$ is honest then receive from $\mathcal{A}$, new configuration $(\mathcal{P}'', t''_a, t''_p, t''_f)$ and reconstruct the $n'-1$ sharings to compute $h_q^k$, $h_e^k$ and thereby, $[z_j^k]_d = h_q^k(0).h_e^k(0) - h_q^k(0).g_b^k(\alpha_j) - h_e^k(0).g_a^k(\alpha_j) + g_c^k(\alpha_j)$ shares of corrupt parties $P_j \in (P_a \cup P_p)$. If $P_{king}$ is actively corrupted, then, for $k \in \{1, ..., l\}$, simulator forwards the received elements to $\mathcal{A}$. Receive from $\mathcal{A}$ new configuration and shares of corrupted parties. Send the computed shares and new configuration received from $\mathcal{A}$ to $\mathcal{F}_{Multiplication}$. Heartbeat towards $P_{king}$ can be simulated since all its inputs are known to the simulator.
3. Validate consistency for $P_{king}$ : Since the elements $q, e$ are public, simulator continues to be honest by forwarding the shares of corrupt parties to the adversary $\mathcal{A}$ while executing this step. If the output of Fault Detection step during execution of CheckConsistencyKing is "happy" then for $k \in \{1, ..., l\}$, send shares of corrupted parties in $P'$ from the above

execution generated by $\mathcal{A}$ to $\mathcal{F}_{Multiplication}$. For corrupted parties not in $P'$ send random shares to $\mathcal{F}_{Multiplication}$. Otherwise continue to the next step after Fault Detect. If the parties do not agree on set $E$ then abort else send (ACTIVESET,$E$) or (CRASHSET,$E$) from execution to $\mathcal{F}_{Multiplication}$.

4. Rechecking the reconstructions : Simulator computes the values of $q^1, ..., q^l$ and $e^1, ..., e^l$. Simulator already knows the shares held by corrupt parties of $[a^k]_{n'-1}$ and $[b^k]_{n'-1}$ for $k \in \{1, ..., l\}$ from generate multiplication triplets phase. It also knows the shares held by corrupt parties for $[x^k]_d$ and $[y^k]_d$. Simulator already sent random values for reconstruction of $[q^k]_{n'-1}$ and $[e^k]_{n'-1}$ on behalf of honest parties. Hence, it learns all shares of $[q^k]_{n'-1}$ and $[e^k]_{n'-1}$. Simulator randomly computes $d-$ sharings of $[q^k]_d$, $[e^k]_d$ of honest parties from its knowledge of corrupt shares of $[q^k]_d$, $[e^k]_d$ and $q^k, e^k$. After the execution of this step, it forwards the configuration received from the adversary to $\mathcal{F}_{Multiplication}$.
5. Rechecking the randomness from multiplication triplets of party $P_i$ :
Let $P_{check}$ be set of actively and passively corrupt parties that check $(n'-1)$ sharings when TripletShareRandom was invoked by all parties. $d_{check} = |P_{check}| \leq min(2t'_a, t'_a + t'_p)$. Simulator sets the shares of $[s^k]_{n'-1}$ held by honest parties in set $\mathcal{H}' = n' - (n' - 2t'_a - t'_p - min(t'_a, t'_p)) - t'_a - t'_p - d_{check}$ to random uniform values. Let $\mathcal{H} - \mathcal{H}'$ represent rest of the honest parties. By the property of Hyper-Invertible matrix, there exists a matrix $N$ such that $(s_i^k)_{P_i \in \mathcal{H}-\mathcal{H}'} = N(a_i^{n'-(n'-2t'_a-t'_p-min(t'_a,t'_p))+1}, ..., a_i^{n'}, (a_i^k)_{P_k \in P_{check}}, (s_i^k)_{P_k \in P_a \cup P_p \cup \mathcal{H}'})$
where $(s_i^k)_{P_i \in \mathcal{H}-\mathcal{H}'}$ represents vector of shares held by the rest of the honest parties,
$(a_i^{n'-(n'-2t'_a-t'_p-min(t'_a,t'_p))+1}, ..., a_i^{n'}, (a_i^k)_{P_k \in P_{check}}, (s_i^k)_{P_k \in P_a \cup P_p \cup \mathcal{H}'})$ is the vector containing shares of $a_i$ held by the parties, shares held by the corrupt parties involved in verification phase i.e. $P_{check}$ of first elements of $k$ multiplication triplets and the shares of $[s^k]_{n'-1}$ held by honest parties in $\mathcal{H}'$ and corrupt parties in $(P_a \cup P_p)$.
We have $(s_i^k)_{P_i \in \mathcal{H}-\mathcal{H}'} = N(q_i^1, ..., q_i^l, (a_i^k)_{P_j \in P_{check}}, (s_i^k)_{P_k \in P_a \cup P_p \cup \mathcal{H}'})$ - $N(x_i^1, ..., x_i^l, 0, ..., 0)$.
Let $U_i = N(x_i^1, ..., x_i^l, 0, ..., 0)$ which is $([U^j]_d)_{P_j \in \mathcal{H}-\mathcal{H}'} =$
$N([x^1]_d, ..., [x^l]_d, 0, ..., 0)$. This makes $U_i$ as the vector of $i$th shares of $([U^j]_d)_{P_j \in \mathcal{H}-\mathcal{H}'}$. Hence, $s_i^j = V_i^j - U_i^j$ where
$V_i^j = N(q_i^1, ..., q_i^l, (a_i^k)_{P_j \in P_{check}}, (s_i^k)_{P_k \in P_a \cup P_p \cup \mathcal{H}'})$.
The shares of $[s^k]_{n'-1}$ held by corrupt parties are already fixed. Simulator also knows $U_i^j$ for actively and passively corrupt parties, that is the linear combination of $i-$th shares of $[x^1]_d, ..., [x^l]_d$, for corrupt parties and sets $V_i^j = s_i^j + U_i^j$.
Further, simulator generates $[_1s^j]_d, [_2s^j]_d, [_3s^j]_d$ honestly for honest parties in $\mathcal{H}'$ and $[_1V^j]_d, [_2V^j]_d, [_3V^j]_d$ are randomly generated for rest of the honest parties in $\mathcal{H} \setminus \mathcal{H}'$ such that for $k \in \{1, 2, 3\}$ and $P_m \in \mathcal{P}_k$,

the $m-$th share of $[_kV^j]_d$ is $V_m^j$ and set $[_ks^j]_d = [_kV^j]_d - [U^j]_d$ for $k \in \{1,2,3\}$. Simulator learns all shares of $[_ks^j]_d = [_kV^j]_d - [U^j]_d$ for $k \in \{1,2,3\}$ which are held by corrupt parties. For every honest party, simulator also sends $m-$th share of $[_ks^j]_d$ to $P_m$ for all $P_m \in (P_a \cup P_p) \setminus \mathcal{P}_k$ to make up the rest of the matrix.
Simulator forwards the generated shares of corrupt parties to $\mathcal{F}_{Multiplication}$ on behalf of adversary $\mathcal{A}$.

6. Check 4-Consistency : Simulator honestly emulates $\mathcal{F}_{4-Consistency}$. If Simulator receives set $E$ or $P_{faultyset}$ as output from adversary $\mathcal{A}$ during the emulation of $\mathcal{F}_{4-Consistency}$ then set $E$ is forwarded to $\mathcal{F}_{Multiplication}$, if received ACTIVESET then forward (ACTIVESET,E) else received CRASHSET then forward (CRASHSET,E) to $\mathcal{F}_{Multiplication}$. Otherwise simulator forwards shares of corrupt parties $([_0r^j]_n, [_1r^j]_n, [_2r^j]_n, [_3r^j]_n)$ for corrupt party $P_n$.
Receive all shares of corrupt parties of all random 4-consistent tuples. For every corrupt party $P_n$, $([_0u^j]_n, [_1u^j]_n, [_2u^j]_n, [_3u^j]_n)$. Send these shares to adversary $\mathcal{A}$.
If Simulator receives set $E$ or $P_{faultyset}$ as output from adversary $\mathcal{A}$ then set $E$ is forwarded to $\mathcal{F}_{Multiplication}$, if received ACTIVESET then forward (ACTIVESET,E) else received CRASHSET then forward (CRASHSET,E) to $\mathcal{F}_{Multiplication}$. Otherwise, send OK to $\mathcal{F}_{Multiplication}$.
7. Finding disputed parties : Simulator randomly generates $[_jq^{i^*}]_d$ such that $_jq_k^{i^*}$ is the $k-$th share of $[q^{i^*}]_{n'-1}$ for honest party $P_k \in \mathcal{P}_j$ and $[x^{i^*}]_d + M_{i^*}([_js^1]_d, ..., [_js^{n'}]_d)$ for corrupt parties. Send the corrupt party shares to $\mathcal{F}_{Multiplication}$.

We need to show that the output distribution is identical in both, real and ideal worlds. Simulator generates $[_jq^{i^*}]_d$ for $j = 1,2,3$ in step 7. Here, we prove that its distribution is identical in real and ideal world. If the size of $|\mathcal{P}_j \cup P_a \cup P_p| \geq d+1$ then all shares of $[_jq^{i^*}]_d$ can be determined by corrupt parties. For $j = 1,2,3$ and corrupt parties $P_j$, the $k-th$ share of $[_jq^{i^*}]_d$ is fixed in both worlds by the adversary$\mathcal{A}$. Similarly, for honest parties $P_k \in \mathcal{P}_j$ the $k-th$ share of $[_jq^{i^*}]_d$ is the $k-th$ share of $[q^{i^*}]_{n'-1}$ set by honest parties in both worlds. Consider $[_jq^{i^*}]_d = [x^{i^*}]_d + M_{i^*}([_js^1]_d, ..., [_js^{n'}]_d)$, where $[_js^k]_d]$ is a random $d-$ sharing such that for every honest $P_i \in \mathcal{P}_j$, $_js_i^k$ is the $i-th$ share of $[s^k]_{n'-1}$ ensured by $\mathcal{F}_{4-Consistency}$. Also, for the corrupt parties, share $_js_i^k$ is already sent by corrupt parties $P_i \in (P_a \cup P_p)$. Hence, $[_js^k]_d]$ is a random $d-$ sharing when the shares of corrupt parties in $\mathcal{P}_j$ are fixed by adversary, thereby making $[_jq^{i^*}]_d$ a random $d-$ sharing. This makes the distribution of $[_jq^{i^*}]_d$ for $j = 1,2,3$ identical for both worlds. The same proof follows for $[_je^{i^*}]_d$ with $j = 1,2,3$.

Simulator generates $[_1s^i]_d, [_2s^i]_d, [_3s^i]_d$ in step 5. Here, we prove that its distribution is identical in both, real and ideal worlds. Since it involves $[s^1]_{n'-1}, [s^2]_{n'-1}, ..., [s^{n'}]_{n'-1}$, we first ensure that the view of its shares is uniformly random. For an honest party $P_j$, the shares obtained by corrupted parties of $[s^j]_{n'-1}$ are fixed by honest party. Consider an honest party $P_i$ who

received the shares that satisfy $(a_i^1, .., a_i^{n'}) = M(s_i^1, .., s_i^{n'})$ during execution of TripletShareRandom. For $j \in \{n' - l + 1, ..., n'\}$, $a_i^j$ is fixed by correctness of TripletShareRandom and computed as $a_i^j = [q_i^j]_{n'-1} - [x_i^j]_d$. For corrupt parties $P_j$, $s_i^j$ is fixed as an honest party $P_i$ receives it from a corrupt party. Also, if $j \leq \{n' - l\}$ and $P_j$ is a corrupt party whose shares of $[a^j]_{n'-1}$ are checked during TripletShareRandom, $a_i^j$ is fixed since it has been sent by a corrupt party. Hence, $(n' - 2t'_a) + t'_a + t'_{a_check} \leq n'$ number of fixed values of $(a_i^1, .., a_i^{n'}), (s_i^1, .., s_i^{n'})$ are fixed. This makes the distribution of $\{s_i^j\}$ for honest parties $P_j$ uniformly random and satisfying $(a_i^1, .., a_i^{n'}) = M(s_i^1, .., s_i^{n'})$. Due to property of Hyper-Invertible Matrix, using $(n' - 2t'_a) + t'_a + t'_{a_check}$ elements of $\{s_i^j\}$ for honest $P_j$, simulator can construct the remaining values for honest $P_j$. Now, simulator chose a set $\mathcal{H}'$ of $(n' - (n' - 2t'_a) - t'_a - t'_{a_check})$ honest parties and set the value $s_i^j$ uniformly random for $P_j \in \mathcal{H}'$ in step 5. It simulates the parties in $\mathcal{H}'$ by generating all shares of $[s^j]_{n'-1}$ honestly. In case of rest of the honest parties, simulator genrates the shares of $[s^j]_{n'-1}$ held by corrupted parties only. For the rest of the honest parties $P_j$, using $(a_i^1, ..., a^(n - 2t'_a)_i, (a_i^j)_{P_j \in P_{check}}, (s_i^j)_{P_j \in P_a \cup P_p \cup \mathcal{H}'})$, $\{s_i^j\}$ are computed. This makes the distribution of $\{[s^j]_{n'-1}\}$ for honest party $P_j$ identical in both the real and ideal world. Also, $[_1s^i]_d, [_2s^i]_d, [_3s^i]_d$ have same distribution in both worlds because they are computed using $\{[s^j]_{n'-1}\}$ for honest party $P_j$.
Complexity : In step 2, the communication complexity of *GenerateMultiplicationtriplets*$(d, d', n' - 1, l, (P', t'_a, t'_p, t'_f))$. is $O(ln\kappa + n^2\kappa + BA(\kappa))$. In step 3, for each multiplication $P_{king}$ receives and sends $O(n\kappa)$ bits.
In step 4, *CheckConsistencyKing* is performed twice where the communication complexity is $O(n\kappa + (t_a + t_f)(n^2\kappa + BA(\kappa)))$.
In step 5, *PubReconRobust* is also performed twice with the communication complexity (public reconstruction complexity $n^2\kappa$).
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 $O(2n\kappa + (t_a + t_f)(2n^2\kappa + BA(\kappa)))$.
In step 8, all parties send $O(n)$ elements to $P_{king}$ and broadcast from $P_{king}$ along with heartbeat takes $O(BA(\kappa))$.
Hence, the communication complexity of $c_m$ multiplication gates is $O(c_m n\kappa + \frac{c_m}{l}(t_a + t_f)(2n^2\kappa + BA(\kappa)))$.

### 4.5 Functionality $\mathcal{F}_{MPC}$

We provide a simple functionality of $\mathcal{F}_{MPC}$ which evaluates a circuit inputted by parties in $P$. It handles the case of fail corrupt parties whose inputs are $\Lambda$ by considering the default input as 0. It also provides set of parties with input $\Lambda$ to output receiving parties which now know if a party participated in the protocol.

**Functionality $\mathcal{F}_{MPC}$**

1. Receive a circuit $C$ from each party and inputs to $C$ from involved parties.
2. If $C$ cannot be evaluated or different values for $C$ are received then execute Complete Break Down.
3. Set the inputs for circuit $C$ to 0 for parties whose input was $\Lambda$.
4. Evaluate $C$ and send its output and set of parties whose input was $\Lambda$ to the respective parties from output gate.

### 4.6 Realizing functionality $\mathcal{F}_{MPC}$

**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 5: 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 7.** ***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 $O(c_i(n\kappa + BA(\kappa))$*

*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 $O(c_i(n\kappa + BA(\kappa))$

**Generating Random 0-Sharings** ZeroShareRandom is yet another protocol that was described by Goyal et al. in [14] which has been inherited in this work. This protocol generates $l$ random t-sharings of 0. Initially, parties generates $l$ random triplet sharings of $r$ using a polynomial of degree $d'$, such that $([a]_{d'}, [b]_{d'}, [c]_{d'})$ are the triplet 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 6: ZeroShareRandom**$(l, (P', t'_a, t'_p, t'_f))$

1. Distributing random triplet sharings : Every party $P_i \in P'$ performs *TripletShareRandom*$(d', d', d', l, (P', t'_a, t'_p, t'_f))$ where $d' = t'_a + t'_p$ to generate $[r^{n'-l+1}]_{d',d',d'}, ..., [r^{n'}]_{d',d',d'}$. If *TripletShareRandom*$(d', d', d', l, (P', t'_a, t'_p, t'_f))$ outputs disputed parties then all parties halt.
2. Local computation : Every triplet 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 8.** ***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 $O(ln'\kappa + n'^2\kappa)$*

*Proof.* Correctness and Secrecy : Since *ZeroShareRandom* invokes *TripletShareRandom*, the correctness and secrecy follows from lemma 3.2. Complexity : Since the protocol is based on $TripletShareRandom$, the complexity $O(ln'\kappa + n'^2\kappa)$ follows from lemma 3.2.

**Output Gate**

**Protocol 7: 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 9.** ***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 $O((n + \lceil \frac{c_o}{l} \rceil)(c_o n\kappa + ln\kappa + n^2\kappa))$*

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

**Realizing $\mathcal{F}_{MPC}$ in the ($\mathcal{F}_{Preparation}, \mathcal{F}_{Multiplication}-$) Hybrid Model** We realize the functionality $\mathcal{F}_{MPC}$ with the following protocol:

**Protocol 8: MainProtocolFunctionality**$(C)$

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

1. Preparation : Each party $P_i \in P$ sends configuration $(P, t_a, t_p, t_f)$ and value $c_i + c_m$ to $\mathcal{F}_{Preparation}$. Functionality $\mathcal{F}_{Preparation}$ outputs a new configuration set as $(P', t'_a, t'_p, t'_f)$ and each party in $P'$ stores the $c_i + c_m$ number of triplets.
2. Input Gates : Each party $P_i \in P$ performs InputGateEval$(P_i, [a]_d, (P', t'_a, t'_p, t'_f))$ where $[a]_d$ belongs to $[a]_d, [b]_d, [c]_d$ the multiplication triplet corresponding to the $c_i$ input gates. If output is "success" then every other $P_j \in P'$ stores the shares. Otherwise, every $P_j$ adds $P_i$ to a set $D$
3. For each $P_i \in D$, every $P_j \in P'$ sets the share of input gates corresponding to $P_i$ such that the input value was 0.
4. Evaluation : For addition gate, each party $P_i \in P'$ computes it's output share by locally applying the addition operation to the shares. For multiplication gates, 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 $k \in \{1, ..., \lceil \frac{c_m}{l} \rceil\}$, let $([a_i^k]_d, [b_i^k]_d, [c_i^k]_d)$, $([a_i^k]_{n'-1}, [b_i^k]_{n'-1}$ and $[x_i^k]_d, [y_i^k]_d$ be $d$-sharings, $(n'-1)$-sharings of multiplication triplet and $d$-sharings of gate inputs respectively. Each player $P_i \in P'$ sends $l$, $(P', t'_a, t'_p, t'_f)$, $([a_i^k]_d, [b_i^k]_d, [c_i^k]_d)$, $([a_i^k]_{n'-1}, [b_i^k]_{n'-1}$ and $[x_i^k]_d, [y_i^k]_d$ to functionality $\mathcal{F}_{Multiplication}$. $\mathcal{F}_{Multiplication}$ outputs, for every $P_i \in P'$ a valid configuration $(P'', t''_a, t''_p, t''_f)$ and share of output $[z_i^k]_d$. Every party sets $P' = P'', t'_a = t''_a, t'_p = t''_p, t'_f = t''_f$.
5. Output Gates : Let $o$ be the number of outputs. All parties perform *OutputGateEval*$(o, (P', t'_a, t'_p, t'_f))$.

**Theorem 5.** *Assuming that* $3t_a + 2t_a + t_f < n$*, the protocol MainProtocolFunctionality(C) securly evaluates* $\mathcal{F}_{MPC}$ *in the* $\mathcal{F}_{Preparation}, \mathcal{F}_{Multiplication}-$*hybrid model, in the presence of a static mixed adversary. Also, the communication complexity of* ***MainProtocolFunctionality*** *is* $O((n + \lceil \frac{c_o}{l} \rceil)(c_o n\kappa + ln\kappa + n^2\kappa))$

*Proof.* We construct a simulator $S_{MPC}$ towards real-world adversary $\mathcal{A}$. It interacts with ideal-world functionality with a black-box access to $\mathcal{A}$ and executing the protocol. We have to prove that output is distributed correctly in both worlds.

**Simulator** $\mathcal{S}_{MPC}$

Simulator $\mathcal{S}_{MPC}$ outputs whatever $\mathcal{A}$ outputs.

1. Preparation : Emulating $\mathcal{F}_{Preparation}$: Send configuration $(P, t_a, t_p, t_f)$ and value $l$ to $\mathcal{A}$.
   Receive valid configuration $(P', t'_a, t'_p, t'_f)$ and shares $(([a_j^k]_d, [b_j^k]_d, [c_j^k]_d))_{k=1,\ldots,l}$ and $(([a_j^k]_{n'-1}, [b_j^k]_{n'-1}))_{k=1,\ldots,l}$ held by $P_j$ $in P_a \cup P_p$. Set $D = \emptyset$.
2. Input Gates : For each input gate with respect to party $P_i \in P'$,
   If the party $P_i$ is actively or passively corrupted then choose a polynomial $g_a$ randomly from a set of polynomials with degree $d$, such that $g_a$ passes through the points in set $\{(\alpha_j, [a_j]_d) | P_j \in (P_a \cup P_p)\}$ where $[a_j]_d$ is the corrupted share of party $P_j \in (P_a \cup P_p)$ from multiplication triplet $[a_j]_d, [b_j]_d, [c_j]_d$ corresponding to input gate. All inputs to the PrivReconRobust, and subsequently, Heartbeat and broadcast protocol in InputGateEval are set. Emulate InputeGateEval. Otherwise, only party $P_i$ receives messages during PrivReconRobust which is not a passively or actively corrupted party, emulate InputGateEval with random $e$ as input to broadcast.
   If the result of InputGateEval is "success", save the corrupted shares as $a_j + e'$ where $e'$ was output of broadcast. Otherwise, the corrupted shares are set to 0.
   If $P_i$ is fail-corrupted and output of InputGateEval is "fail" then add $P_i$ to set $D$.
   If $P_i$ was actively corrupted then if the result of InputGateEval is "success" then compute the input by evaluating $e' + g_a(0)$. Otherwise, consider input as $\Lambda$.
   After evaluating all the input gates for $P_i \in P'$, set the corrupted shares of output to 0 if $P_i \in D$.
3. Evaluation : For addition gate, each corrupt party $P_j \in (P_a \cup P_p)$ computes it's output share by locally applying the addition operation to the shares it holds.
   Emulating $\mathcal{F}_{Multiplication}$ :

For each segment $k \in \{1, ..., \lceil \frac{c_m}{l} \rceil\}$, let $([a_j^k]_d, [b_j^k]_d, [c_j^k]_d)$, $([a_j^k]_{n'-1}, [b_j^k]_{n'-1}$ and $[x_j^k]_d, [y_j^k]_d$ be $d$-sharings, $(n'-1)$-sharings of multiplication triplet and $d$-sharings of gate inputs respectively held by corrupt parties $P_j \in P_a \cup P_p$ respectively.
For each $k = 1, ..., \lceil \frac{c_m}{l} \rceil$, choose $q^k$ and $e^k$ randomly on behalf of corrupt parties to compute $[q_j^k]_{n'-1} = [x_j^k]_d + [a_j^k]_{n'-1} + q^k$ and $[e_j^k]_{n'-1} = [y_j^k]_d + [b_j^k]_{n'-1} + e^k$. choose a polynomial $h_q^k, h_e^k$ randomly from a set of polynomials with degree $n'-1$, such that $h_q^k$ passes through the points in set $\{(\alpha_j, [q_j]_d) | P_j \in (P_a \cup P_p)\}$ and point $(0, q^k)$, & $h_e^k$ passes through the points in set $\{(\alpha_j, [e_j]_d) | P_j \in (P_a \cup P_p)\}$ and point $(0, e^k)$.
Send to adversary $l$, $(P', t'_a, t'_p, t'_f)$, for $k \in \{1, ..., l\}$, $h_q^k, h_e^k$ and $[a_j^k]_d, [b_j^k]_d, [c_j^k]_d, [a_j^k]_{n'-1}, [b_j^k]_{n'-1}$.
Receive from adversary a configuration $(P'', t''_a, t''_p, t''_f)$ and $[z_j^k]_d$ for $P_j \in (P_a \cup P_p)$ i.e. corrupted shares of output $z^k$. Set $P' = P'', t'_a = t''_a, t'_p = t''_p, t'_f = t''_f$.

4. Output : In simulation, set the inputs of actively-corrupt parties and fail-corrupt parties(set it to $\Lambda$) to the OutputGateEval and receive the outputs of actively and passively corrupted parties.
Foe all parties $P_i$ which receive the output value, if $P_i \in P_a \cup P_p$ then let $[s_j]_d$ such that $P_j \in P_a \cup P_p$ be the corrupted shares of input and $s$ is the output received from functionality $\mathcal{F}_{MPC}$. Choose a polynomial $g_s$ randomly from a set of polynomials with degree $d$, such that $g_s$ passes through the points in set $\{(\alpha_j, [s_j]_d) | P_j \in (P_a \cup P_p)\}$ and point $(0, s)$. Now, simulator can compute all the inputs to OutputGateEval from $g_s$ and emulate towards $\mathcal{A}$.

We have to prove that the simulator interaction with adversary $\mathcal{A}$ is the same as in the real world. For private reconstruction, if a party towards which the value is constructed is not actively or passively corrupted then there is no interaction with corrupt parties. Hence, the simulation is trivial. Otherwise, simulator uses a random shares as input to Private Reconstruction protocol which is same as the real world. For broadcast, only if the sender is actively or passively corrupt then it obtains the input. In this case, simulator can recompute the messages sent by every party. Heartbeat protocol can be emulated by the simulator without any issue since simulator knows the set of crashed parties. Hence, it can be seen that the inputs of the corrupted parties, the corrupted shares of output by simulator(i.e.$e' + g_a(0)$) and set $D$ is same as in the real world.

For addition gates, it is clear to see that the corrupted shares in the simulator are same as in the real world as the operation is local and the corrupted shares of input are already shown to be same as the real world.

For multiplication gate, in real world, the parties have shares of correct $d$ and $n'-1$ multiplication triplets which is ensured by $\mathcal{F}_{Preparation}$ and shares of correct inputs which are ensured by InputGateEval. $\mathcal{F}_{Multiplication}$ returns the

shares of $xy$ where the polynomial selected is consistent with the shares of corrupted parties provided by $\mathcal{A}$. In ideal world, since the output value is $xy$, simulator also chooses polynomial which is consistent with the shares of corrupted parties provided by $\mathcal{A}$.
In real world, adversary receives sharing polynomials $h_q, h_e$ and the corrupted shares of triplets $[a_j]_d, [b_j]_d, [c_j]_d, [a_j]_{n'-1}, [b_j]_{n'-1}$. Since $a$ and $b$ were randomly selected by $\mathcal{F}_{Preparation}$, the values generated by polynomials $h_q, h_e$ i.e. $q = x + a, e = y + e$ are random and independet of adversary view. In ideal world, the shares of corrupted parties are given to the simulator by $\mathcal{F}_{Preparation}$ which selected them randomly which also makes the values generated by polynomials in simulator random. Hence, the distribution of corrupt shares is identical in real and ideal world. Also, the values generated by polynomials $h_q, h_e$ are also random and independent of adversary view in real and ideal world.
In output phase, since the inputs sent by simulator to $\mathcal{F}_{MPC}$ are same as evaluated in InputGateEval in ideal world and the circuit is correctly evaluated in real world, the sharings reconstructed by ReconPrivRobust have same coefficients of random polynomials in both real and ideal worlds. Further, these random polynomials are also consistent with the corrupted shares known to adversary $\mathcal{A}$. This implies that the inputs to ReconPrivRobust are identically distributed in both worlds.
Complexity : For $c_i$ input gates, the communication complexity is $O(c_i(n\kappa + BA(\kappa)))$.
For $c_m$ multiplication gates, the communication complexity is $O(c_m n\kappa + \lceil \frac{c_m}{l} \rceil (t_a + t_f)(2n^2\kappa + BA(\kappa)))$.
For $c_o$ output gates, the communication complexity is $O((n + \lceil \frac{c_o}{l} \rceil)(c_o n\kappa + ln\kappa + n^2\kappa))$.
Hence, the overall communication complexity of evaluation phase is $O((c_i + c_m + c_o)n\kappa + (c_i + c_m)(BA(\kappa)) + n^3k + nBA(\kappa) + {c_o}^2 n\kappa)$.

## 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, Yuval Ishai, and Mikkel Krøigaard. Perfectly secure multiparty computation and the computational overhead of cryptography. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 445–465. Springer, 2010.
9. Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference*, pages 572–590. Springer, 2007.
10. 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.
11. 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.
12. 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.
13. 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.
14. 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.
15. 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.
16. 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.
17. Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In *Annual International Cryptology Conference*, pages 463–482. Springer, 2006.
18. Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
19. Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.