# AD-MPC: Fully Asynchronous Dynamic MPC with Guaranteed Output Delivery

Wenxuan Yu
*202115114@mail.sdu.edu.cn*
*Shandong University*

Minghui Xu
*mhxu@sdu.edu.cn*
*Shandong University*

Bing Wu
*bwu12340@gmail.com*
*Shandong University*

Sisi Duan
*duansisi@tsinghua.edu.cn*
*Tsinghua University*

Xiuzhen Cheng
*xzcheng@sdu.edu.cn*
*Shandong University*

## Abstract

Traditional secure multiparty computation (MPC) protocols presuppose a fixed set of participants throughout the computational process. To address this limitation, Fluid MPC [CRYPTO 2021] presents a dynamic MPC model that allows parties to join or exit during circuit evaluation dynamically. However, existing dynamic MPC protocols can guarantee safety but not liveness within asynchronous networks. This paper introduces $\Pi_{\text{AD-MPC}}$, a fully asynchronous dynamic MPC protocol. $\Pi_{\text{AD-MPC}}$ ensures both safety and liveness with optimal resilience, capable of tolerating $t$ ($n = 3t+1$) corrupted participants. To achieve this, we develop a novel asynchronous transfer protocol $\Pi_{\text{Trans}}$ and a preprocessing protocol $\Pi_{\text{Aprep}}$ specifically tailored for dynamic environments. In contrast to most dynamic MPC protocols that achieve security with abort in synchronous networks, $\Pi_{\text{AD-MPC}}$ guarantees output delivery in asynchronous networks with optimal resilience, thus enhancing robustness. We provide a formal security proof of $\Pi_{\text{AD-MPC}}$ under the Universal Composability (UC) framework. Furthermore, an extensive evaluation involving up to 20 geographically distributed nodes demonstrates the protocol's practical performance and its ability to reliably deliver outputs in asynchronous dynamic settings. Compared to the state-of-the-art Fluid MPC, $\Pi_{\text{AD-MPC}}$ achieves comparable performance while offering significantly enhanced security guarantees.

## 1 Introduction

Secure multiparty computation (MPC) [8, 34, 46] allows a group of servers to collaboratively compute a function while preserving the privacy of their inputs. MPC protocols have been deployed in a variety of applications, such as anonymous voting systems [4], privacy-preserving machine learning [38], and private smart contracts [3, 5]. Traditional MPC protocols typically assume a static set of participants who must commit to participating in the entire circuit evaluation process. However, this assumption prohibits dynamic participant addition or removal, which restricts the deployment of MPC protocols in scenarios involving complex circuits.

Recent works present dynamic MPC to address the issue of dynamic participation. Dynamic MPC divides an arithmetic circuit into multiple layers, each computed by a distinct set of servers (i.e., a committee). Servers intending to partake in the next layer of circuit evaluation invoke an election function, which yields a subset of servers designated as the next committee following the completion of the current layer's evaluation. Fluid MPC [19] is the seminal work in dynamic MPC, introducing the maximal fluidity setting where servers participate in a single communication round per layer. Dynamic MPC protocols achieve security for dishonest majority [12, 44], linear communication complexity [12], and guaranteed output delivery [30].

The dynamic MPC model is well-suited for distributed systems like permissionless blockchains [14, 40] deployed over asynchronous wide-area networks (WANs). In asynchronous networks, there are no timing assumptions and messages can experience arbitrary delays. Unfortunately, existing dynamic MPC protocols [12, 19, 30, 44] can only guarantee safety in asynchronous networks, not liveness. This limitation has motivated us to design and implement a practical dynamic MPC protocol for asynchronous settings.

### 1.1 Challenges and Our Solutions

**Challenge I: How to guarantee liveness?** In the dynamic MPC model, communication occurs during committee handovers. The current committee transfers secret sharings of the current layer's outputs to the next committee. Existing dynamic MPC protocols rely on a synchronous network assumption, where the next committee receives all shares from the current committee within an upper time bound. The next committee then reconstructs new polynomials based on these shares. Each reconstructed polynomial's constant term represents the corresponding gate's output. However, this time assumption doesn't hold in asynchronous networks. The next committee can become stuck waiting indefinitely for all

shares, preventing the protocol from ever terminating. Moreover, the arrival order of the message in honest parties cannot be guaranteed in asynchronous networks. This means that honest parties might reconstruct polynomials based on different, potentially incomplete sets of shares, which compromises the correctness of the overall evaluation.

**Solution:** We design an asynchronous transfer protocol $\Pi_{\mathsf{Trans}}$ to securely transfer shares between committees. $\Pi_{\mathsf{Trans}}$ leverages asynchronous primitives to ensure both liveness and set consistency during handovers. During the committee handover, each current committee member acts as a dealer and utilizes the Asynchronous Complete Secret Sharing (ACSS) protocol [42] to distribute secret shares to the next committee. The ACSS protocol guarantees that valid shares distributed from a dealer all lie on a polynomial. Upon receiving $n-t$ valid ACSS instances, the next committee calls the Multi-valued Validated Byzantine Agreement (MVBA) protocol [15] to reach consensus on a common subset containing at least $n-t$ valid instances. Subsequently, the next committee reconstructs the new polynomial based on the common subset. These steps ensure that all incoming committee members reconstruct the polynomial using the same set of shares, thereby guaranteeing consistency.

**Challenge II: How to achieve guaranteed output delivery with optimal resilience ($n = 3t+1$)?** Beyond safety and liveness, achieving robustness becomes crucial. This property guarantees the correctness of outputs in the presence of a malicious adversary. Existing dynamic MPC protocols (e.g., [12, 19, 44]) achieve security with abort (SwA) in synchronous networks. This means honest parties agree on whether the protocol aborts or continues. However, SwA is not robust. An adversary performing an additive attack [32] on any layer can render the output unavailable. Guaranteed output delivery (GOD) is a stronger notion than SwA and is often synonymous with robustness [39]. GOD ensures that corrupted parties cannot prevent honest parties from receiving correct outputs. To the best of our knowledge, none of the existing MPC protocols achieves GOD in asynchronous dynamic settings. The dynamic MPC protocol introduced in [30] achieves GOD, but it is limited to synchronous networks and struggles with polynomial reconstruction in asynchronous networks. Asynchronous MPC protocols [1, 20–24, 39] achieve GOD under the assumption of static participants. However, these protocols cannot guarantee robustness considering dynamic participants. To achieve GOD with optimal resilience, it is crucial to guarantee the accuracy of the final output by ensuring the correctness of values produced by each committee. The evaluation by each committee involves computation and handover phases, each posing challenges for GOD. During computation, multiplication elevates the polynomial degree from $t$ to $2t$, mandating degree reduction back to $t$. Under optimal resilience, this process lacks robustness [2]. Therefore, avoiding degree-$2t$ polynomial reconstruction poses a challenge. During the handover phase, verifying if transferred values match original secret shares held by current committee servers is another critical problem. Using compromised transferred values for polynomial reconstruction leads to an incorrect polynomial with a constant term deviating from the gate's output.

**Solution:** We propose two novel improvements to address the aforementioned issues. First, our protocol builds upon the offline/online preprocessing paradigm (see section 3.1 for more details) introduced in [39, 44]. The paradigm eliminates the need for generating degree-$2t$ polynomials during the online phase. However, the generation of Beaver triples in the offline phase, which are consumed in the online phase, still involves degree-$2t$ polynomials. We design an asynchronous dynamic preprocessing protocol $\Pi_{\mathsf{Aprep}}$ to generate Beaver triples without reconstructing degree-$2t$ polynomials. Unlike the dynamic MPC protocol [44] that relies on universal preprocessing, where dynamism is restricted to parties involved in the universal preprocessing phase, we let each committee invoke $\Pi_{\mathsf{Aprep}}$ to dynamically generate Beaver triples for the subsequent committee, thus allowing any server to participate in the evaluation phase.

Second, to prove the transferred value matches the original secret during the handover phase, we leverage the robustness of Shamir secret sharing. Current committee servers collaboratively generate a secret sharing of a random value, locally adding it to a secret share for masked share. An additive homomorphic commitment (such as Pedersen commitment [43]) is used to prove the masked-secret share relationship. Alongside ACSS for secret distribution, the committee broadcasts masked shares and commitments to the next committee using the Asynchronous Reliable Broadcast (RBC) protocol. Subsequent committee servers verify masked-secret share relationships and then use the robustness of Shamir secret sharing to identify verified masked shares reconstructing the masked value. This reconstruction, while preserving secret privacy, establishes a valid dealer set where each dealer's polynomial constant term is their original secret. Interpolating a new polynomial from these dealers' ACSS instances guarantees its constant term aligns with the previous layer's circuit gate output.

## 1.2 Contributions

We design an asynchronous transfer protocol $\Pi_{\mathsf{Trans}}$ to guarantee liveness. We further propose an asynchronous dynamic preprocessing protocol $\Pi_{\mathsf{Aprep}}$ to generate Beaver triples without reconstructing degree-$2t$ polynomials. By integrating $\Pi_{\mathsf{Trans}}$ and $\Pi_{\mathsf{Aprep}}$, we construct the first fully asynchronous dynamic MPC protocol with optimal resilience. The protocol not only guarantees safety and liveness but also achieves GOD. We implement our asynchronous dynamic MPC protocol, which achieves similar performance to Fluid MPC while ensuring GOD. The practical implementation is open-sourced at https://anonymous.4open.science/r/

Anonymous-Authors-ADMPC.

## 2 Related Work

The dynamic MPC model was first introduced by Fluid MPC [19]. Subsequent dynamic MPC protocols have achieved security against dishonest majority [12, 44] and linear communication complexity [12], while ensuring maximal fluidity. All of these dynamic MPC protocols only achieve security with abort (SwA), where honest parties reach a consensus on whether the protocol aborts. DGL23 [30] achieves guaranteed output delivery (GOD) with maximal fluidity, but at the cost of requiring two additional committees to rectify errors within the current layer. Fluid MPC and DGL23 are based on the classic BGW protocol [8], where parties locally multiply their secret shares of two values, then perform a degree reduction step to reduce the degree of the resulting polynomial from $2t$ to $t$. These dynamic MPC protocols improve dynamic MPC in synchronous networks but do not address asynchronous networks.

In an asynchronous network, there are no timing assumptions, and messages can be arbitrarily delayed. We define a protocol as fully asynchronously safe and live if it can ensure both properties in asynchronous networks. All of the aforementioned dynamic MPC protocols only achieve safety in such environments. Asynchronous MPC protocols [1, 20, 22–24, 39] achieve fully asynchronous liveness and safety by combining with asynchronous primitives. HoneybadgerMPC [39] cannot guarantee liveness during the offline phase, limiting it to achieving fully asynchronous safety. COPS16 [21] implements a partial asynchronous setting by introducing a synchronous point under the asynchronous network, thereby not satisfying fully asynchronous liveness. The aforementioned asynchronous MPC protocols [1, 20, 22–24, 39] rely on the robustness of Shamir secret sharing to guarantee GOD. However, these protocols do not consider dynamic participants. With optimal resilience, if the adversary controls $t$ corrupted parties, the protocol's robustness can be compromised by dynamic network conditions, such as node failures.

## 3 Model and Preliminaries

### 3.1 Model

We consider a client-server model in a dynamic MPC setting. In this scenario, a set of clients delegates the computation of a function to a server network. The function is transformed into a layered arithmetic circuit $\tilde{C}$ [19]. Assuming the depth of $\tilde{C}$ is $d$, for each $l \in [d]$, the inputs to the $(l+1)$-th layer circuit $\tilde{C}^{l+1}$ come from the outputs of its previous layer $\tilde{C}^l$ and are evaluated by a server set (i.e., a committee) denoted by $\mathcal{P}^{l+1}$. We divide the evaluation process into three stages: the input stage, the execution stage, and the output stage. During the input stage, clients provide inputs to committee $\mathcal{P}^1$, and during the output stage, they receive outputs from committee $\mathcal{P}^d$. In the execution stage, each committee $\mathcal{P}^l$ evaluates the circuit $\tilde{C}^l$ and passes the output to the next committee $\mathcal{P}^{l+1}$.

*Offline/online preprocessing paradigm.* During the offline phase, parties collaboratively generate Beaver triples $([a],[b],[c])$, where $c = a \cdot b$ and $[a]$, $[b]$, $[c]$ are degree-$t$ Shamir secret sharings. These triples are used during the online phase when evaluating multiplication gates. Given secret sharings of inputs $[x],[y]$, and sharing of a Beaver triple $([a],[b],[c])$, parties locally compute shares of $[\alpha] = [x] - [a]$ and $[\beta] = [y] - [b]$. Then, they broadcast their shares to reconstruct $\alpha$ and $\beta$, from which parties can locally compute shares of $[xy] = \alpha \cdot \beta + [a] \cdot \beta + [b] \cdot \alpha + [c]$.

*Fluidity.* Fluidity is defined as the number of rounds of interaction within a layer [19]. In synchronous networks, existing dynamic MPC protocols can achieve *maximal fluidity* within a single communication round. However, since asynchronous consensus algorithms cannot be completed within one communication round, dynamic MPC cannot achieve maximal fluidity in asynchronous networks.

*Committee election.* The committee election for each layer is dynamic and separate from the circuit evaluation. Given that the committee election methods (e.g., [11, 19, 33, 45]) can seamlessly integrate into our model, we opt not to delve into the specifics of committee election. We focus on ensuring that $\mathcal{P}^l$ can reach consensus on the composition of the next committee $\mathcal{P}^{l+1}$ through committee election. For further insights into committee election, refer to [19].

*Adversary model.* We denote the number of clients as $N_c$, and the committee size as $N_s$. We set $N_c = 3t_c + 1$ and $N_s = 3t_s + 1$, where $t_c$ and $t_s$ are the upper bounds on the number of corrupted parties among the clients and the committee, respectively. For simplicity, we designate $\mathcal{P}^0$ as the client set. We consider an *R-adaptive adversary* as defined in [19]. Specifically, the adversary $\mathcal{A}$ first statically selects corrupted clients before the input stage. During the execution stage, at the beginning of $\tilde{C}^l$ with committee $\mathcal{P}^l$, $\mathcal{A}$ adaptively chooses a subset of servers to corrupt. Once a server is corrupted, $\mathcal{A}$ can retroactively access its entire historical state and send messages on its behalf. Therefore, for $P_i^l$ participating in the evaluation of $\tilde{C}^m$ $(m < l)$, if $\mathcal{A}$ wants to corrupt $P_i^l$ in $\tilde{C}^l$, it succeeds only if the sum of the number of corrupted servers in $\tilde{C}^m$ and $P_i^l$ does not exceed the threshold $t$.

*Network assumption.* We assume an asynchronous network, which means that there are no timing assumptions and messages can be arbitrarily delivered, but all messages sent between honest parties must ultimately be delivered [7].

*Fully asynchronous dynamic MPC.* We summarize the asynchronous dynamic MPC model considered in this work in the following definition.

**Definition 1** (*Fully Asynchronous Dynamic MPC with Guaranteed Output Delivery*)**.** *With optimal resilience ($n = 3t+1$), we say that a dynamic MPC protocol $\pi$ is a fully asynchronous MPC with guaranteed output delivery if it satisfies the follow-*

Table 1: Comparisons of existing dynamic and asynchronous MPC protocols

| Dynamic | $t <$ | Fluidity | SwA[1] | GOD[2] Online | GOD[2] Offline | Under Fully Async Safety | Under Fully Async Liveness | Practical[3] Deployment | Security Goals |
|---|---|---|---|---|---|---|---|---|---|
| Fluid MPC [19] | $n/3$ | $O(1)$ | ✓ | ✗ | - | ✓ | ✗ | ✓ | Statistical |
| Le Mans [44] | $n$ | $O(1)$ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | Statistical |
| DGL23 [30] | $n/3$ | $O(1)$ | ✓ | ✓ | - | ✓ | ✗ | ✗ | Perfect[4] |
| BEP23 [12] | $n$ | $O(1)$ | ✓ | ✗ | ✗ | ✓ | ✗ | ✗ | Statistical |
| Asynchronous | | | | | | | | | |
| HoneyBadgerMPC [39] | $n/3$ | - | ✓ | ✓ | ✗ | ✓ | ✗ | ✓ | Perfect |
| CHP13 [20] | $n/4$ | - | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | Statistical |
| CP15 [22] | $n/3$ | - | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | Computational |
| CP16 [23] | $n/3$ | - | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | Computational |
| COPS16 [21] | $n/2$ | - | ✓ | ✓ | ✓ | ✓ | ✗ | ✗ | Computational |
| CP23 [24] | $n/3$ | - | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | Statistical |
| AAPP24 [1] | $n/4$ | - | ✓ | ✓ | ✓ | ✓ | ✓ | ✗ | Perfect |
| **This work** | $n/3$ | $O(\log n)$ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | Computational |

[1] SwA denotes security with abort.
[2] GOD denotes guaranteed output delivery.
[3] Those MPC protocols that have been validated through experimental deployment.
[4] DGL23 [30] achieves perfect security at $t < n/3$ and computational security at $t < n/2$ while achieving guaranteed output delivery.

*ing properties:*

- **Safety (Privacy):** The R-adaptive adversary $\mathcal{A}$ cannot infer more private information than what is already known to honest clients and servers, based on the information obtained from corrupted servers and clients.
- **Liveness:** The protocol $\pi$ will eventually terminate.
- **Guaranteed Output Delivery (Robustness):** The R-adaptive adversary $\mathcal{A}$ should not be able to prevent honest parties from receiving correct outputs [37]. Specifically, let $d$ be the depth of the circuit. For $l \in [d]$, honest servers in $\mathcal{P}^l$ can correct errors injected by $\mathcal{A}$ from previous committee $\mathcal{P}^{l-1}$. Honest clients can correct errors from corrupted servers in $\mathcal{P}^d$, ensuring the correctness of the outputs.

## 3.2 Dynamic MPC

We summarize the following two key technical aspects of the dynamic MPC model.

**Transfer protocol.** A transfer protocol in dynamic MPC is designed to securely transfer sharings to the next committee. These protocols leverage the re-randomization phase of the BGW protocol [8] for transfer. Let $[z^l]$ denote the output of a gate in $\tilde{C}^l$, where $[z^l]$ is a degree-$t$ polynomial. Each server $P_i^l$ in $\mathcal{P}^l$ generates a degree-$t$ polynomial $[z_i^l]$, with its constant term being the secret share $z_i^l$ of $z^l$. For each $P_j^{l+1}$ in $\mathcal{P}^{l+1}$, $P_i^l$ sends the corresponding share $(j, z_{i,j}^l)$ to $P_j^{l+1}$. Upon receiving secret sharings $\{[z_1^l], \cdots, [z_n^l]\}$ from the previous committee, servers in $\mathcal{P}^{l+1}$ select a common subset $T$ of size $|T| = t+1$. They then perform Lagrange interpolation on the sharings in $T$ to reconstruct a degree-$t$ polynomial. Consequently, $\mathcal{P}^{l+1}$ obtains a new polynomial whose constant term is the original secret $z^l$. The idea of multiplication evaluation is similar to the transfer protocol. Servers locally multiply shares and then utilize the transfer protocol to distribute the output shares to the next committee. However, a slightly larger common subset size ($|T| = 2t+1$) is used for reconstruction in this stage.

**Security with abort in dynamic MPC.** To achieve security against the R-adaptive adversary, dynamic MPC employs a consolidated check approach [31, 32, 41], where the correctness of the computation (for the entire circuit) is checked once. These protocols leverage MAC techniques [10, 27] to commit to their private inputs. During the input stage, clients generate sharings of a random MAC key $[r]$, along with additional randomness. In the execution stage, for each input sharing $[z]$, an additional committee $\mathcal{P}^1$ computes a secret sharing of a MAC on it. The following committee utilizes additional randomness to aggregate the outputs of each circuit gate. let $[u]$ and $[v]$ denote the aggregate sharing and the corresponding MAC, respectively. During the output stage, clients verify the validity of all the MACs in one shot. They achieve this by locally computing $[T] = [v] - [r] \cdot [u]$ and then reconstructing $[T]$ to check whether it equals 0. If $T = 0$, the

evaluation is considered valid, and clients obtain the output. Otherwise, clients abort the protocol, discarding the result.

### 3.3 Asynchronous Primitives

Here, we present the asynchronous primitives used in our protocol. We refer readers to Appendix D for a formal description of the security of these ideal functionalities $\mathcal{F}_{\mathsf{ACSS}}$, $\mathcal{F}_{\mathsf{RBC}}$ and $\mathcal{F}_{\mathsf{MVBA}}$.

**Asynchronous Complete Secret Sharing (ACSS).** An ACSS protocol guarantees agreement among parties on the successful completion of the sharing phase, even if the dealer is corrupted. This protocol should satisfy the following properties when operating in an asynchronous network [28, 29]:

- *Correctness:* If the dealer $L$ is honest, then every honest party $P_i$ will eventually output a share $p(i)$ during the sharing phase, where $p(\cdot)$ is a random degree-$t$ polynomial with $p(0) = s$. Upon completion of the sharing phase, if all honest parties invoke the reconstruction phase, they will output $s$ as long as at most $t$ parties are corrupted.
- *Secrecy:* If $L$ is honest and all honest parties have not yet begun executing the reconstruction phase, then an adversary that corrupts up to $t$ parties has no information about $s$.
- *Completeness:* If some honest party terminates the sharing phase, then there exists a degree $t$ polynomial $p(\cdot)$ over $\mathbb{Z}_q$ such that $p(0) = s'$ and each honest party $P_i$ will eventually hold a share $s_i = p(i)$. Moreover, when $L$ is honest, $s' = s$.

**Asynchronous Reliable Broadcast (RBC).** Informally, an RBC protocol allows a designated sender to broadcast an input message to all parties in asynchronous networks, even in the presence of a malicious adversary $\mathcal{A}$. An RBC protocol satisfies the following properties [39]:

- *Validity:* If the sender is honest and inputs $m$, then all honest parties deliver $m$.
- *Agreement:* If any two honest parties deliver $m$ and $m'$, then $m = m'$.
- *Totality:* If any honest party delivers $m$, then all honest parties deliver $m$.

**Multi-valued Validated Byzantine Agreement (MVBA).** An MVBA protocol [15] allows a set of parties, where each party provides an input value, to eventually agree on the same value that satisfies a predefined external predicate $P(m) : \{0,1\}^{|m|} \to \{0,1\}$, which is known to all parties. The MVBA protocol satisfies the following properties [26, 36]:

- *Termination:* If every honest party inputs with an externally valid value, then every honest party outputs a value.
- *External-Validity:* If an honest party outputs a value $m$, then $P(m) = 1$.
- *Agreement:* All honest parties that terminate output the same value.
- *Integrity:* If all parties are honest and if some parties output $m$, them some parties proposed $m$.

## 4 Technical Overview

### 4.1 Achieving Liveness in Asynchronous Dynamic MPC

**Key Technique I: Achieve consistent polynomial interpolation across committee handovers.** In section 3.2, we introduce the transfer protocol for the dynamic MPC model. However, this protocol is not well-suited for asynchronous networks. Due to the lack of guaranteed synchronization, the parties cannot definitively agree on the set of received shares. If parties utilize Lagrange interpolation based on inconsistent share sets, the reconstructed points will not be distributed over the same polynomial. Consequently, these points cannot be used to reconstruct the polynomial whose constant term is the original secret.

To ensure polynomial consistency among committee members in an asynchronous network, we utilize $\mathcal{F}_{\mathsf{ACSS}}$ and $\mathcal{F}_{\mathsf{MVBA}}$ to construct the asynchronous transfer protocol $\Pi_{\mathsf{Trans}}$. Specifically, servers in $\mathcal{P}^l$ first invoke $\mathcal{F}_{\mathsf{ACSS}}$ to distribute their shares. The functionality $\mathcal{F}_{\mathsf{ACSS}}$ ensures that only ACSS instances distributed on a degree-$t$ polynomial will be received by $\mathcal{P}^{l+1}$. Verified ACSS instances are then collected into the local set $LT_i$ held by every server $P_i^{l+1}$. Once $|LT_i| = 2t+1$, $P_i^{l+1}$ invokes $\mathcal{F}_{\mathsf{MVBA}}$ to agree on a common subset $T$, from which the first $t+1$ shares are selected to interpolate a new polynomial. $\Pi_{\mathsf{Trans}}$ guarantees liveness in asynchronous networks by leveraging $\mathcal{F}_{\mathsf{ACSS}}$ and $\mathcal{F}_{\mathsf{MVBA}}$. Furthermore, $\Pi_{\mathsf{Trans}}$ guarantees polynomial consistency during committee handovers.

### 4.2 Achieving GOD with Optimal Resilience

**Key Technique II: Avoid reconstructing degree-$2t$ polynomials during the offline phase.** As discussed earlier, achieving GOD with optimal resilience necessitates avoiding the reconstruction of degree-$2t$ polynomials. However, traditional MPC protocols employing "double sharing" [6] for Beaver triple generation still reconstruct degree-$2t$ polynomials. In double sharing, a random value $r$ is shared using both $t$- and $2t$-sharings. Servers then calculate $\langle c \rangle = [a] \cdot [b]$, where $[a]$

and $[b]$ are uniformly random sharings, resulting in a degree-$2t$ polynomial $\langle c \rangle$. Subsequently, servers locally compute $\langle \gamma \rangle = \langle c \rangle - \langle r \rangle$ and broadcast. After reconstructing $\gamma$, servers can compute $[c] = \gamma + [r]$ to get the triple $([a],[b],[c])$. The outlined process for generating Beaver triples necessitates the reconstruction of a degree-$2t$ polynomial $\langle \gamma \rangle$, compromising the robustness under optimal resilience.

We design an asynchronous dynamic preprocessing protocol $\Pi_{\mathsf{Aprep}}$ that dynamically generates triples without reconstructing degree-$2t$ polynomials. Specifically, let's assume $\tilde{C}^{l+1}$ has only one multiplication gate. Server $P_i^l$ (for $i \in [n]$) in $\mathcal{P}^l$ first generates two types of triples $([a_i^l],[b_i^l],[c_i^l])$ and $([x_i^l],[y_i^l],[z_i^l])$, with the latter triple used to verify $c_i^l = a_i^l \cdot b_i^l$. $P_i^l$ then calls $\mathcal{F}_{\mathsf{ACSS}}$ to distribute shares of these triples to $\mathcal{P}^{l+1}$. Servers in $\mathcal{P}^{l+1}$ gather the verified triples $([a_i^l],[b_i^l],[c_i^l])$ and call $\mathcal{F}_{\mathsf{MVBA}}$ on it to output a common subset $|T|$, where $|T| = 2t+1$. $\mathcal{P}^{l+1}$ selects the first $t+1$ $[a_i^l]$ and $[b_i^l]$ from $T$ to interpolate new degree-$t$ polynomials $U(\cdot)$ and $V(\cdot)$. To construct a degree-$2t$ polynomial $W(\cdot) = U(\cdot) \cdot V(\cdot)$, we note that the first $t+1$ $[c_i^l]$ are already distributed on $W(\cdot)$. Hence, we require $t$ additional points on $W(\cdot)$. $\mathcal{P}^{l+1}$ identifies points $i$ (for $i \in \{t+2, \cdots 2t+1\}$) on $U(\cdot)$ and $V(\cdot)$, and use another $t$ triples from $T$ to compute $W(i) = U(i) \cdot V(i)$. This process yields $2t+1$ correct points for interpolating $W(\cdot)$, guaranteeing that any point $j$ on $U(\cdot)$, $V(\cdot)$, and $W(\cdot)$ satisfies $W(j) = U(j) \cdot V(j)$.

**Key Technique III: Guarantee robustness in $\Pi_{\mathsf{Trans}}$.** The protocol $\Pi_{\mathsf{Trans}}$ ensures that committee members interpolate the polynomial based on a consistent share set. However, it cannot guarantee that the constant term of the interpolated polynomial matches the original secret. Although $\mathcal{F}_{\mathsf{ACSS}}$ confirms that ACSS instances originate from a degree-$t$ polynomial generated by the dealer, a corrupted dealer could produce a polynomial whose constant term does not match the initially held secret. To achieve guaranteed output delivery, $\mathcal{P}^l$ must prove to $\mathcal{P}^{l+1}$ that the constant of the polynomial generated is indeed the output of $\tilde{C}^l$.

We achieve GOD using the commitment ideal functionality $\mathcal{F}_{\mathsf{Commit}}$ (see Appendix D for details) with additive homomorphic property. Specifically, assuming $\tilde{C}^l$ contains only one gate. Each server $P_i^l$ (for $i \in [n]$) receives a random share $\alpha_i^l$ from the previous committee $\mathcal{P}^{l-1}$ and locally computes a masked share $m_i^l = z_i^l + \alpha_i^l$, where $z_i^l$ is its shares of the circuit output. $P_i^l$ invokes $\mathcal{F}_{\mathsf{Commit}}$ to generate a commitment for its random share, then calls $\mathcal{F}_{\mathsf{RBC}}$ to broadcast its masked share and commitment to $\mathcal{P}^{l+1}$. Meanwhile, $P_i^l$ calls $\mathcal{F}_{\mathsf{ACSS}}$ to distribute its secret share like in the original $\Pi_{\mathsf{Trans}}$. Each server $P_j^{l+1}$ utilizes the additive homomorphic property of $\mathcal{F}_{\mathsf{Commit}}$ to verify the relationship between the masked share and the ACSS instance. Then, $P_j^{l+1}$ puts the verified masked share to a local set $LT_j$. Upon $|LT_j| \geq 2t+1$, $P_j^{l+1}$ invokes online error correction $\mathsf{OEC}$ (see Appendix C for details) to find a set $GT_j$ from $LT_j$ where the polynomial interpolated from $2t+1$ masked shares satisfies $p(0) = m^l$. Since $[\alpha^l]$ is uniformly random and independent, reconstructing the masked value $m^l$ does not reveal the original secret $z^l$. Subsequently, $P_j^{l+1}$ invokes $\mathcal{F}_{\mathsf{MVBA}}$ to output a common subset $T$. Using the masked value, we demonstrate (without revealing the original secret) that the secret shares corresponding to ACSS instances from $T$ are all distributed in a degree-$t$ polynomial. Finally, $P_j^{l+1}$ interpolates a new polynomial based on the first $t+1$ ACSS instances in $T$.

# 5 Protocol Design

## 5.1 AD-MPC Protocol

As shown in Figure 1, our asynchronous dynamic MPC protocol $\Pi_{\mathsf{AD\text{-}MPC}}$ is divided into three main stages:

**Input Stage:** During this stage, clients in $\mathcal{P}^0$ provide their private inputs and additional randomness for circuit evaluation to the first committee $\mathcal{P}^1$. Specifically, each client $P_i^0$ (for $i \in [n]$) calls $\mathcal{F}_{\mathsf{ACSS}}$ to distribute private input $s_i$. Concurrently, $P_i^0$ invokes $\Pi_{\mathsf{Rand}}$ and $\Pi_{\mathsf{Aprep}}$ to generate random values $([\alpha_1^1],...,[\alpha_n^1])$ and Beaver triples $\{([a_1^1],[b_1^1],[c_1^1]),\cdots,([a_{c_m}^1],[b_{c_m}^1],[c_{c_m}^1])\}$ for $\mathcal{P}^1$, where $c_m$ denotes the number of multiplication gates in the subsequent layer.

**Execution Stage:** This stage is responsible for circuit evaluation, carried out by committees composed of servers. Given a layered arithmetic circuit $\tilde{C}$, with a depth of $d$, each committee $\mathcal{P}^l$ (for $l \in [d]$) is responsible for the evaluation of $\tilde{C}^l$. $\mathcal{P}^l$ first evaluates the current circuit, which includes the following types of gates:

- **Addition:** To perform $[z^l] = [x^l] + [y^l]$, $P_i^l$ locally computes $z_i^l = x_i^l + y_i^l$.
- **Addition by Constant:** To perform $[z^l] = [x^l] + c$, $P_i^l$ locally computes $z_i^l = x_i^l + c$.
- **Multiplication by Constant:** To perform $[z^l] = c \cdot [x^l]$, $P_i^l$ locally computes $z_i^l = c \cdot x_i^l$.
- **Multiplication:** To perform $[z^l] = [x^l] \cdot [y^l]$, servers in $\mathcal{P}^l$ run $\Pi_{\mathsf{Mult}}$ with $([x^l],[y^l])$ and triple $([a^l],[b^l],[c^l])$ as inputs to get sharing $[z^l]$.

Subsequently, servers in $\mathcal{P}^l$ (for $l \in [d-1]$) invoke $\Pi_{\mathsf{Rand}}$ and $\Pi_{\mathsf{Aprep}}$ to generate random values and Beaver triples for the next committee $\mathcal{P}^{l+1}$. Finally, $\mathcal{P}^l$ (for $l \in [d-1]$) calls $\Pi_{\mathsf{Trans}}$ with $[\alpha_i^l]$ and $[z_i^l]$ (for $i \in [w]$) as inputs to securely transfer the output to the next committee $\mathcal{P}^{l+1}$, where $w$ denotes the width of $\tilde{C}^l$.

**Output Stage:** After completing the evaluation of $\tilde{C}^d$, $\mathcal{P}^d$ invokes $\Pi_{\mathsf{Trans}}$ to transfer the final output of the circuit to $\mathcal{P}^0$. Upon receiving these shares, clients in $\mathcal{P}^0$ call $\Pi_{\mathsf{Rec}}$ to reconstruct the output.

The workflow of our protocol is shown in Figure 2.

Protocol $\Pi_{\mathsf{AD\text{-}MPC}}$

(A) **Input Stage**

1. Each client $P_i^0$ (for $i \in [n]$) in $\mathcal{P}^0$ calls $\mathcal{F}_{\mathsf{ACSS}}$ with $s_i$ as input to generate sharing of $[s_i]$.
2. Clients in $\mathcal{P}^0$ run $\Pi_{\mathsf{Rand}}$ with $n$ as input to generate random values $([\alpha_1^1], ..., [\alpha_n^1])$ for the next committee $\mathcal{P}^1$.
3. Clients in $\mathcal{P}^0$ run $\Pi_{\mathsf{Aprep}}$ with $c_m$ as input to generate Beaver triples $\{([a_1^1], [b_1^1], [c_1^1]), \cdots, ([a_{c_m}^1], [b_{c_m}^1], [c_{c_m}^1])\}$ for $\mathcal{P}^1$, where $c_m$ is the number of multiplication gates on the next layer.

(B) **Execution Stage**

1. Servers in $\mathcal{P}^l (l \in [d])$ evaluates the current layer:
   - Addition: To perform $[z^l] = [x^l] + [y^l]$, $P_i^l$ locally computes $z_i^l = x_i^l + y_i^l$.
   - Addition by Constant: To perform $[z^l] = [x^l] + c$, $P_i^l$ locally computes $z_i^l = x_i^l + c$.
   - Multiplication by Constant: To perform $[z^l] = c \cdot [x^l]$, $P_i^l$ locally computes $z_i^l = c \cdot x_i^l$.
   - Multiplication: To perform $[z^l] = [x^l] \cdot [y^l]$, servers in $\mathcal{P}^l$ run $\Pi_{\mathsf{Mult}}$ with $([x^l], [y^l])$ and $([a^l], [b^l], [c^l])$ as inputs to get sharing $[z^l]$.
2. Servers in $\mathcal{P}^l$ (for $l \in [d-1]$) run $\Pi_{\mathsf{Rand}}$ with $w$ as input to generate random value $([\alpha_1^{l+1}], ..., [\alpha_w^{l+1}])$ for the next committee $\mathcal{P}^{l+1}$, where $w$ is the width of the next layer.
3. Servers in $\mathcal{P}^l$ (for $l \in [d-1]$) run $\Pi_{\mathsf{Aprep}}$ with $c_m$ as input to generate Beaver triples $\{([a_1^{l+1}], [b_1^{l+1}], [c_1^{l+1}]), \cdots, ([a_{c_m}^{l+1}], [b_{c_m}^{l+1}], [c_{c_m}^{l+1}])\}$, where $c_m$ is the number of multiplication gates on the next layer.
4. Servers in $\mathcal{P}^l$ (for $l \in [d-1]$) run $\Pi_{\mathsf{Trans}}$ with $[\alpha_i^l]$ and $[z_i^l]$ (for $i \in [w]$) as inputs to securely transfer sharing $[z_i^l]$ to the next committee $\mathcal{P}^{l+1}$.

(C) **Output Stage**

1. Servers in $\mathcal{P}^d$ run $\Pi_{\mathsf{Trans}}$ with $[\alpha_i^d]$ and $[z_i^d]$ (for $i \in [w]$) as inputs to securely transfer sharing $[z_i^d]$ to $\mathcal{P}^0$.
2. Upon receiving sharing $[z_i^d]$ from $\mathcal{P}^d$, clients in $\mathcal{P}^0$ run $\Pi_{\mathsf{Rec}}$ with $[z_i^d]$ as input to reconstruct value $z_i^d$ and output.

Figure 1: Asynchronous Dynamic MPC

## 5.2 Building Blocks

This section describes the building blocks required by $\Pi_{\mathsf{Aprep}}$ and $\Pi_{\mathsf{Trans}}$.

*Generating random values.* We design the protocol $\Pi_{\mathsf{Rand}}$ to generate random values. The basic idea is that $\mathcal{P}^l$ employs the randomization extraction technique RE (see Appendix A for details) based on the hyper-invertible matrix to generate a batch of random values for $\mathcal{P}^{l+1}$. The hyper-invertible property of this matrix ensures that we can extract $n-t$ uniformly random and independent secret sharings from $n$ inputs.

Protocol $\Pi_{\mathsf{Rand}}$

- **Input:** The number of random values $w$.
- **Output:** The shares of random values $\{[r_1^{l+1}], ..., [r_w^{l+1}]\}$.
- **Procedure:**

**Committee $\mathcal{P}^l$:**

1. If $w > n-t$, servers in $\mathcal{P}^l$ call $\Pi_{\mathsf{Rand}}$ $\lfloor \frac{w}{n-t} \rfloor$ times using $n-t$ as input, and then call $\Pi_{\mathsf{Rand}}$ again with an input of $w - \lfloor \frac{w}{n-t} \rfloor$. Otherwise, continue with the following steps.
2. Each server $P_i^l$ (for $i \in [n]$) chooses a random element $e_i^l \leftarrow \mathbb{Z}_q$.
3. $P_i^l$ invokes $\mathcal{F}_{\mathsf{ACSS}}$ with $e_i^l$ as input to $\mathcal{P}^{l+1}$.

**Committee $\mathcal{P}^{l+1}$:**

4. Request output from $\mathcal{F}_{\mathsf{ACSS}}$. After receiving $n-t$ valid request-based delayed ACSS instances $(e_{i,j}^l, \pi_{e_i^l})$ from $P_i^l$, $P_j^{l+1}$ (for $i \in [n]$) designates these valid instances as $T_j$ and maintains a set $S_j$ to record all received valid instances. Define predicate $P(\cdot)$ outputs 1 only when $|T_j| = n-t$ and $T_j \in S_j$. $P_j^{l+1}$ calls $\mathcal{F}_{\mathsf{MVBA}}$ with $(T_j, S_j, P(\cdot))$ as input.
5. Request output form $\mathcal{F}_{\mathsf{MVBA}}$ until receiving $T$, where $|T| = n-t$. Servers in $\mathcal{P}^{l+1}$ set $[e_i^l] = 0$ for $i \notin T$.
6. Servers in $\mathcal{P}^{l+1}$ calls RE with their shares as inputs to obtain random values $\{[r_1^{l+1}], ..., [r_w^{l+1}]\}$.

Figure 3: Protocol $\Pi_{\mathsf{Rand}}$

Specifically, each server $P_i^l$ (for $i \in [n]$) initially selects a random value $e_i^l \leftarrow \mathbb{Z}_q$, where $\mathbb{Z}_q$ is a finite field. Then, $P_i^l$ calls $\mathcal{F}_{\mathsf{ACSS}}$ with $e_i^l$ as input to $\mathcal{P}^{l+1}$. Upon receiving valid ACSS instances through $\mathcal{F}_{\mathsf{ACSS}}$, $P_j^{l+1}$ adds these instances to a set $T_j$. Let the set $S_j$ record all valid instances. For a set $T_k$ from $P_k^{l+1}$, define an external predicate $P(\cdot)$ that outputs 1 if and only if $|T_k| = 2t+1$ and $T_k \in S_j$. When $|T_j| = 2t+1$, $P_j^{l+1}$ calls $\mathcal{F}_{\mathsf{MVBA}}$ with $(T_j, S_j, P(\cdot))$ as input. Let $T$ be the output of $\mathcal{F}_{\mathsf{MVBA}}$, determines which $e_{i,j}^l$ values are considered valid, setting $e_{i,j}^l = 0$ for any $i \notin T$. Servers in $\mathcal{P}^{l+1}$ locally run RE using a public $w$-by-$n$ Vandermonde matrix to randomize the extraction from the seed, The protocol $\Pi_{\mathsf{Rand}}$ is shown in Figure 3.

*Robust reconstruction.* The primary function of $\Pi_{\mathsf{Rec}}$ is to robustly reconstruct a secret from shares provided by servers. Specifically, servers in $\mathcal{P}^l$ calls $\mathcal{F}_{\mathsf{RBC}}$ to broadcast their shares. Upon receiving at least $2t+1$ shares, $P_i^l$ (for $i \in [n]$) uses these

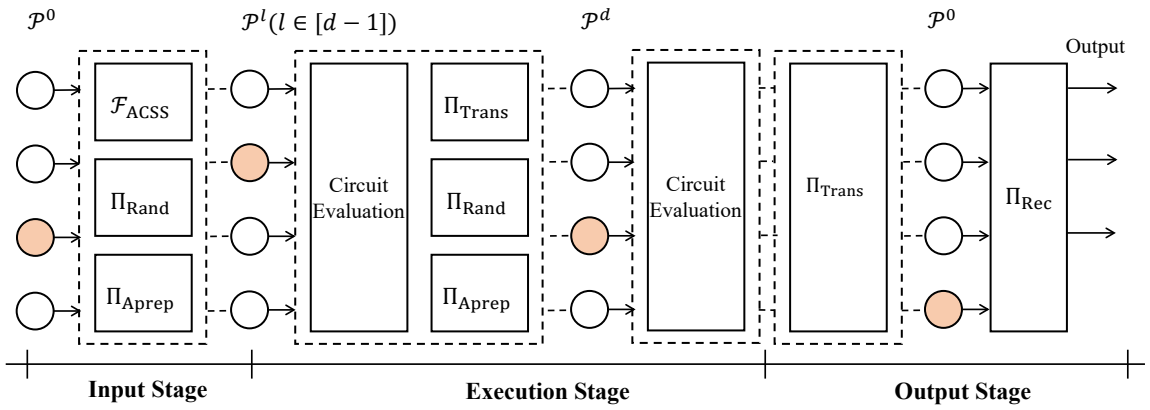



Figure 2: The workflow of $\Pi_{\text{AD-MPC}}$.

shares to invoke OEC function, till the secret is reconstructed. The protocol is depicted in Figure 4.

**Protocol $\Pi_{\text{Rec}}$**

- **Input:** The shares $[x^l]$.
- **Output:** The value $x^l$.
- **Procedure:**
  **Committee $\mathcal{P}^l$:**
  1. Server $P_i^l$ (for $i \in [n]$) calls $\mathcal{F}_{\text{RBC}}$ with $x_i^l$ as input.
  2. Request output from $\mathcal{F}_{\text{RBC}}$. $P_i^l$ continuously receives shares from $\mathcal{F}_{\text{RBC}}$ while invoking OEC with these shares as inputs, till $x^l$ is reconstructed.

Figure 4: Protocol $\Pi_{\text{Rec}}$

*Multiplication evaluation.* We design protocol $\Pi_{\text{Mult}}$ to evaluate multiplication during the execution stage. Specifically, when $\mathcal{P}^l$ evaluates a multiplication gate with inputs $[x^l]$, $[y^l]$, and a Beaver triple $([a^l],[b^l],[c^l])$. Each server $P_i^l$ (for $i \in [n]$) locally computes $\gamma_i^l = x_i^l - a_i^l$ and $\delta^l = y_i^l - b_i^l$. Subsequently, $\mathcal{P}^l$ invokes $\Pi_{\text{Rec}}$ to robustly reconstruct $\gamma$ and $\delta$. Once reconstructed, $\mathcal{P}^l$ calculates the the output $[z^l]$ of the multiplication gate using $[z^l] = [c^l] + \gamma^l[b^l] + \delta^l[a^l] + \gamma^l\delta^l$. The protocol $\Pi_{\text{Mult}}$ is shown in Figure 5.

**Protocol $\Pi_{\text{Mult}}$**

- **Input:** Sharing of inputs $[x^l]$ and $[y^l]$, and a Beaver triple $([a^l],[b^l],[c^l])$.
- **Output:** Sharing of multiplication $[z^l]$.
- **Procedure:**
  **Committee $\mathcal{P}^l$:**
  1. Each server $P_i^l$ locally computes $\gamma_i^l = x_i^l - a_i^l$ and $\delta_i^l = y_i^l - b_i^l$.
  2. Servers in $\mathcal{P}^l$ run $\Pi_{\text{Rec}}$ twice, once with $[\gamma^l]$ and once with $[\delta^l]$ as inputs, to robustly reconstruct $\gamma^l$ and $\delta^l$.
  3. Servers in $\mathcal{P}^l$ locally compute $[z^l] = [c^l] + \gamma^l[b^l] + \delta^l[a^l] + \gamma^l\delta^l$ to determine the shares of the multiplication $[z^l]$.

Figure 5: Protocol $\Pi_{\text{Mult}}$

## 5.3 Asynchronous Dynamic Preprocessing

Our asynchronous dynamic preprocessing protocol $\Pi_{\text{Aprep}}$ aims to dynamically generate triples for the following committee, while avoiding the reconstruction of degree-$2t$ polynomials. $\Pi_{\text{Aprep}}$ consists of the following four main stages:

*Sharing random Beaver triples.* The first step in $\Pi_{\text{Aprep}}$ is to share random beaver triples. $P_i^l$ (for $i \in [n]$) first selects two sets of random triples:

$$\{(a_{(i,g)}^l, b_{(i,g)}^l, c_{(i,g)}^l), (x_{(i,g)}^l, y_{(i,g)}^l, z_{(i,g)}^l)\}_{g \in [c_m]}$$

where $c_{(i,g)}^l = a_{(i,g)}^l \cdot b_{(i,g)}^l$, $z_{(i,g)}^l = x_{(i,g)}^l \cdot y_{(i,g)}^l$ and $c_m$ is the number of multiplication gates for $\tilde{C}^{l+1}$. We utilize the standard "sacrificing trick" [24, 27], which involves sacrificing the latter triple to verify that indeed $a_{(i,g)}^l \cdot b_{(i,g)}^l = c_{(i,g)}^l$. $P_i^l$ calls $\mathcal{F}_{\text{ACSS}}$ with these triples as inputs to $\mathcal{P}^{l+1}$. Then servers in $\mathcal{P}^l$ run $\Pi_{\text{Rand}}$ with $n \cdot c_m$ as inputs to generate random values $[r_i^l]$ for $\mathcal{P}^{l+1}$.

Upon receiving valid ACSS instances and random sharings, since $\mathcal{F}_{\text{ACSS}}$ ensures that these shares are distributed over degree-$t$ polynomials, $\mathcal{P}^{l+1}$ no longer needs to validate these instances but needs to verify that $a_{(i,g)}^l \cdot b_{(i,g)}^l = c_{(i,g)}^l$. Specially, each server $P_j^{l+1}$ runs $\Pi_{\text{Rec}}$ to reconstruct $r_i^l$. $P_j^{l+1}$ subsequently locally computes $\rho_{(i,g),j}^l = r_i^l \cdot a_{(i,g),j}^l - x_{(i,g),j}^l$ and $\sigma_{(i,g),j}^l = b_{(i,g),j}^l - y_{(i,g),j}^l$. ‘ Servers in $\mathcal{P}^{l+1}$ run $\Pi_{\text{Rec}}$ to reconstruct $\rho_{(i,g)}^l$ and $\sigma_{(i,g)}^l$. $P_j^{l+1}$ locally computes $\tau_{(i,g),j}^l$, followed by executing $\Pi_{\text{Rec}}$ to reconstruct $\tau_{(i,g)}^l$.

*Agreeing on a Common Subset.* If $\tau_{(i,g)}^l = 0$ for $g \in [c_m]$, it proves that indeed $a_{(i,g)}^l \cdot b_{(i,g)}^l = c_{(i,g)}^l$ without revealing any private information. $P_j^{l+1}$ puts index $i$ into a set $T_j$. Upon receiving $2t+1$ valid triples, let the set $S_j$ record all valid shares. For a set $T_k$ provided by $P_k^{l+1}$, define an external predicate

Protocol $\Pi_{\mathsf{Aprep}}$

- **Input:** The number of multiplication gates $c_m$.
- **Output:** The random Beaver triples $\{([u_1^{l+1}],[v_1^{l+1}],[w_1^{l+1}]),\cdots,([u_{c_m}^{l+1}],[v_{c_m}^{l+1}],[w_{c_m}^{l+1}])\}$
- **Procedure:**

**Sharing Random Beaver Triples:**
**Committee $\mathcal{P}^l$:**

1. Each server $P_i^l$ (for $i\in[n]$) chooses random triples $\{(a_{(i,g)}^l,b_{(i,g)}^l,c_{(i,g)}^l)\}_{g\in\{1,\cdots c_m\}}$, where $c_{(i,g)}^l=a_{(i,g)}^l\cdot b_{(i,g)}^l$.
2. $P_i^l$ chooses random triples $\{(x_{(i,g)}^l,y_{(i,g)}^l,z_{(i,g)}^l)\}_{g\in\{1,\cdots c_m\}}$, where $z_{(i,g)}^l=x_{(i,g)}^l\cdot y_{(i,g)}^l$.
3. $P_i^l$ calls $\mathcal{F}_{\mathsf{ACSS}}$ with $(a_{(i,g)}^l,b_{(i,g)}^l,c_{(i,g)}^l)$ and $(x_{(i,g)}^l,y_{(i,g)}^l,z_{(i,g)}^l)$ as inputs to $\mathcal{P}^{l+1}$.
4. Servers in $\mathcal{P}^l$ run $\Pi_{\mathsf{Rand}}$ with $n$ as inputs to generate random values $[r_i^{l+1}]$ (for $i\in[n]$).

**Committee $\mathcal{P}^{l+1}$:**

5. Request output from $\mathcal{F}_{\mathsf{ACSS}}$. Upon receiving valid shares $\{(a_{(i,g),j}^l,b_{(i,g),j}^l,c_{(i,g),j}^l),(x_{(i,g),j}^l,y_{(i,g),j}^l,z_{(i,g),j}^l),r_{i,j}^{l+1}\}_{i\in[n],g\in[c_m]}$, $P_j^{l+1}$ runs $\Pi_{\mathsf{Rec}}$ to reconstruct $r_i^{l+1}$.
6. $P_j^{l+1}$ locally computes $\rho_{(i,g),j}^{l+1}=r_i^{l+1}\cdot a_{(i,g),j}^l-x_{(i,g),j}^l$, $\sigma_{(i,g),j}^{l+1}=b_{(i,g),j}^l-y_{(i,g),j}^l$.
7. Servers in $\mathcal{P}^{l+1}$ run $\Pi_{\mathsf{Rec}}$ to reconstruct $\rho_{(i,g)}^{l+1}$ and $\sigma_{(i,g)}^{l+1}$.
8. $P_j^{l+1}$ locally computes $\tau_{(i,g),j}^{l+1}=r_i^{l+1}\cdot c_{(i,g),j}^l-z_{(i,g),j}^l-\sigma_{(i,g)}^{l+1}\cdot x_{(i,g),j}^l-\rho_{(i,g)}^{l+1}\cdot y_{(i,g),j}^l-\rho_{(i,g)}^{l+1}\cdot\sigma_{(i,g)}^{l+1}$.
9. Servers in $\mathcal{P}^{l+1}$ run $\Pi_{\mathsf{Rec}}$ to reconstruct $\tau_{(i,g)}^{l+1}$.

**Agreeing on a Common Subset:**
**Committee $\mathcal{P}^{l+1}$:**

10. If $\tau_{(i,g)}^{l+1}=0$ for $g\in\{1,\cdots c_m\}$, $P_j^{l+1}$ puts index $i$ into a set $T_j$. Upon receiving $n-t$ valid indexes, define predicate $P(\cdot)$ outputs 1 only when $|T_j|=n-t$ and $T_j\in S_j$. $P_j^{l+1}$ calls $\mathcal{F}_{\mathsf{MVBA}}$ with $(T_j,S_j,P(\cdot))$ as input.
11. Request output form $\mathcal{F}_{\mathsf{MVBA}}$ until receiving $T$, where $|T|=n-t$.

**Extracting Random Polynomials:**
**Committee $\mathcal{P}^{l+1}$:**

12. $P_j^{l+1}$ sets $u_{(i,g),j}^{l+1}=a_{(i,g),j}^l$, $v_{(i,g),j}^{l+1}=b_{(i,g),j}^l$ and $w_{(i,g),j}^{l+1}=c_{(i,g),j}^l$, with $i=\{1,\cdots,t+1\}$.
13. $P_j^{l+1}$ locally computes $u_{(k,g),j}^{l+1}=\mathsf{Lagrange}(t+1,\{(1,u_{(1,g),j}^{l+1}),\cdots,(t+1,u_{(t+1,g),j}^{l+1})\},k)$ and $v_{(k,g),j}^{l+1}=\mathsf{Lagrange}(t+1,\{(1,v_{(1,g),j}^{l+1}),\cdots,(t+1,v_{(t+1,g),j}^{l+1})\},k)$ with $k=\{t+2,\cdots,2t+1\}$.
14. $P_j^{l+1}$ computes $d_{(i,g),j}^{l+1}=u_{(i,g),j}^{l+1}-a_{(i,g),j}^l$, $e_{(i,g),j}^{l+1}=v_{(i,g),j}^{l+1}-b_{(i,g),j}^l$ with $i=\{t+2,\cdots,2t+1\}$.
15. Servers in $\mathcal{P}^{l+1}$ run $\Pi_{\mathsf{Rec}}$ to reconstruct $d_{(i,g)}^{l+1}$ and $e_{(i,g)}^{l+1}$.
16. $P_j^{l+1}$ locally computes $w_{(i,g),j}^{l+1}=d_{(i,g)}^{l+1}\cdot e_{(i,g)}^{l+1}+d_{(i,g)}^{l+1}\cdot b_{(i,g),j}^l+e_{(i,g)}^{l+1}\cdot a_{(i,g),j}^l+c_{(i,g),j}^l$, with $i=\{t+2,\cdots,2t+1\}$.

**Output Random Triples:**
**Committee $\mathcal{P}^{l+1}$:**

17. For $g\in[c_m]$, $P_j^{l+1}$ locally computes $u_{g,j}^{l+1}=\mathsf{Lagrange}(t+1,\{(1,u_{(1,g),j}^{l+1}),\cdots,(t+1,u_{(t+1,g),j}^{l+1})\},\beta)$, $v_{g,j}^{l+1}=\mathsf{Lagrange}(t+1,\{(1,v_{(1,g),j}^{l+1}),\cdots,(t+1,v_{(t+1,g),j}^{l+1})\},\beta)$ and $w_{g,j}^{l+1}=\mathsf{Lagrange}(2t+1,\{(1,w_{(1,g),j}^{l+1}),\cdots,(2t+1,w_{(2t+1,g),j}^{l+1})\},\beta)$, where $\beta\in\mathbb{Z}_q\backslash\{1,\cdots,2t+1\}$ is a non-zero value.
18. Servers in $\mathcal{P}^{l+1}$ output $\{([u_g^{l+1}],[v_g^{l+1}],[w_g^{l+1}])\}_{g\in[c_m]}$.

Figure 6: Protocol $\Pi_{\mathsf{Aprep}}$

$P(\cdot)$ that outputs 1 if and only if $|T_k|=2t+1$ and $T_k\in S_j$. $P_j^{l+1}$ calls $\mathcal{F}_{\mathsf{MVBA}}$ with $(T_j,S_j,P(\cdot))$ as input. Let $T$ be the output of $\mathcal{F}_{\mathsf{MVBA}}$, and $|T|=2t+1$. From this, we agree on a common subset $T$, consisting of $2t+1$ valid multiplication triples.

*Extracting Random Triples.* Since each triple within the set $T$ is generated independently by potentially corrupted servers in $\mathcal{P}^l$, $T$ may contain triples generated by such servers. These triples are used to evaluate the multiplication, which could leak private information. Therefore, we need additional steps to extract uniformly random and independent triples from the set $T$.

Inspired by [9, 20, 24], we use elements from the set $T$ to interpolate three sets of polynomials: $U_g(\cdot)$, $V_g(\cdot)$ and $W_g(\cdot)$ for $g\in[c_m]$, where $W_g(\cdot)=U_g(\cdot)\cdot V_g(\cdot)$. Both $U_g(\cdot)$ and $V_g(\cdot)$ are degree-$t$ polynomials, while $W_g(\cdot)$ is a degree-$2t$ polynomial. $P_j^{l+1}$ locally sets $u_{(i,g),j}^{l+1}=a_{(i,g),j}^l$, $v_{(i,g),j}^{l+1}=b_{(i,g),j}^l$ and $w_{(i,g),j}^{l+1}=c_{(i,g),j}^l$, with $i\in\{1,\cdots,t+1\}$. Interpolating $U_g(\cdot)$ and $V_g(\cdot)$ can be done directly through the first $t+1$ points, but $W_g(\cdot)$ is a degree-$2t$ polynomial, needs $2t+1$ distinct points to interpolate the polynomial. Therefore, we sacrifice additional $t$ triples from $T$ to evaluate $W_g(i)=U_g(i)\cdot V_g(i)$ with $i\in\{t+2,\cdots,2t+1\}$. Specially, using Lagrange inter-

polation Lagrange (see Appendix B for details), servers in $\mathcal{P}^{l+1}$ can obtain their respective shares of points on the polynomials $U_g(\cdot)$ and $V_g(\cdot)$. For $i \in \{t+2, \cdots, 2t+1\}$, let $u^{l+1}_{(i,g)}$ and $v^{l+1}_{(i,g)}$ represent point $i$ on $U_g(i)$ and $V_g(i)$. Afterwards, servers in $\mathcal{P}^{l+1}$ consume additional $t$ triples from $T$ to compute $w^{l+1}_{(i,g)} = u^{l+1}_{(i,g)} \cdot v^{l+1}_{(i,g)}$. Given that $T$ contains at most $t$ triples from $\mathcal{A}$, $\mathcal{A}$ cannot uniquely interpolate degree-$t$ polynomials $U_g(\cdot)$ and $V_g(\cdot)$. As a result, we obtain three uniformly random polynomials $U_g(\cdot)$, $V_g(\cdot)$ and $W_g(\cdot)$.

*Output Random Triples.* For $g \in [c_m]$, servers in $\mathcal{P}^{l+1}$ invoke Lagrange on point $\beta$ to get triples $(u^{l+1}_g, v^{l+1}_g, w^{l+1}_g)$, where $\beta$ is a predefined value known by servers, set to $2t+2$ for simplicity. By executing this four-step process, $\mathcal{P}^{l+1}$ successfully acquires the necessary triples for $\tilde{C}^{l+1}$. The workflow of $\Pi_{\mathsf{Aprep}}$ is illustrated in Figure 6.

## 5.4 Asynchronous Committee Handovers

To ensure consistent polynomial interpolation and guaranteed output delivery in asynchronous networks, we design the protocol $\Pi_{\mathsf{Trans}}$. The detailed process of $\Pi_{\mathsf{Trans}}$ is shown in Figure 7.

*Robust Secure Transfer.* Let $[z^l_k]$ represents the output of the $k$-th gate in $\tilde{C}^l$. Upon receiving random sharing $[\alpha^l_k]$ from $\Pi_{\mathsf{Rand}}$, each server $P^l_i$ (for $i \in [n]$) locally computes masked share $m^l_{k,i} = z^l_{k,i} + \alpha^l_{k,i}$, where $z^l_{k,i}$ and $\alpha^l_{k,i}$ are shares of $[z^l_k]$ and $[\alpha^l_k]$ held by $P^l_i$. $P^l_i$ calls $\mathcal{F}_{\mathsf{ACSS}}$ with $z^l_{k,i}$ as input to $\mathcal{P}^{l+1}$. Meanwhile, $P^l_i$ runs $\mathcal{F}_{\mathsf{Commit}}$ with $\alpha^l_{k,i}$ to generate commitment $\pi_{\alpha^l_{k,i}}$. Then $P^l_i$ runs $\mathcal{F}_{\mathsf{RBC}}$ with $(m^l_{k,i}, \pi_{\alpha^l_{k,i}})$ to $\mathcal{P}^{l+1}$.

Upon receiving a valid ACSS instance $(z^l_{(k,i),j}, \pi_{z^l_{k,i}})$ from $P^l_i$ through $\mathcal{F}_{\mathsf{ACSS}}$, where $z^l_{(k,i),j}$ represents the share sent to $P^{l+1}_j$ of secret $z^l_{(k,i)}$, and $\pi_{z^l_{k,i}}$ is the commitment to $z^l_{(k,i)}$. $P^{l+1}_j$ invokes $\mathcal{F}_{\mathsf{Commit}}$ with $(\pi_{z^l_{k,i}}, \pi_{\alpha^l_{k,i}}, m^l_{k,i})$ as input. The goal of this step is to check the relationship between the masked share $m^l_{k,i}$ and the secret share $z^l_{k,i}$ without revealing $z^l_{k,i}$, i.e., $m^l_{k,i} = z^l_{k,i} + \alpha^l_{k,i}$. We can implement the above steps using the Pedersen commitment [43] and the ACSS protocol [28] based on it. If $\mathcal{F}_{\mathsf{Commit}}$ outputs 1, $P^{l+1}_j$ adds $i$ to a set $LT_j$. When $|LT_j| = 2t+1$, $P^{l+1}_j$ locally runs OEC to identify a set $GT_j$ from $LT_j$ that contains $2t+1$ masked shares that interpolate a degree-$t$ polynomial $p_j(\cdot)$ satisfying $p_j(0) = m^l_k$. Since the random sharing $[\alpha^l_k]$ is uniformly random and independent, reconstructing the masked value $m^l_k$ does not reveal the original secret $z^l_k$. Moreover, since our previous steps have proven the relationship between the masked share and the secret share, every honest server $P^{l+1}_j$ reconstructing $m^l_k$ with $GT_j$ also implies that the original secret $z^l_k$ can be reconstructed with the secret shares corresponding to the masked shares in that set as well. The next step is to agree on a common subset. $P^{l+1}_j$ maintains the set $LT_j$ to record all received valid shares. For a set $GT_k$ provided by $P^{l+1}_k$, define an external predicate $P(\cdot)$ that outputs 1 if and only if $|GT_k| = 2t+1$ and $GT_k \in LT_j$. $P^{l+1}_j$ calls $\mathcal{F}_{\mathsf{MVBA}}$ with $(GT_j, LT_j, P(\cdot))$ as input. Let $T$ be the output of $\mathcal{F}_{\mathsf{MVBA}}$, servers select the first $t+1$ elements from the set $T$ to form the common subset $CS$. $P^{l+1}_j$ runs Lagrange based on $CS$ to get a new share $z^{l+1}_{k,j}$ on $[z^{l+1}_k]$ that satisfies $z^l_k = z^{l+1}_k$.

# 6 Implementation and Evaluation

## 6.1 Implementation

We implement a prototype of $\Pi_{\mathsf{AD\text{-}MPC}}$ using over 5000 lines of Python code. We use bls12381[1] for elliptic curve operations and asyncio[2] for asynchronous operations. Our prototype supports the evaluation of arbitrary layered arithmetic circuits over a field $\mathbb{Z}_q$. Since Fluid MPC [19] is not open-source, we implement it to facilitate subsequent comparison.

In our implementation, we implement $\mathcal{F}_{\mathsf{Commit}}$ and implement the remaining ideal functionalities by referring to DXKR23 [28] to optimize overhead. DXKR23 relies on the Pedersen commitment [43] to implement its ACSS protocol, achieving a communication overhead of $O(\kappa n^2)$, where $\kappa$ represents the security parameter. Therefore, DXKR23 achieves computational security, and our protocol inherits this security property from DXKR23. If we use a statistically secure ACSS protocol [24] to instantiate $\mathcal{F}_{\mathsf{ACSS}}$, our protocol can achieve statistical security, but at the cost of increased communication overhead to $O(\kappa n^4 + n^5)$. Furthermore, given the Pedersen commitment is additively homomorphic, we leverage it to instantiate $\mathcal{F}_{\mathsf{Commit}}$, which satisfies the Verify function.

For the committee election, similar to Fluid MPC, we omit the election function, allowing $\mathcal{P}^l$ to directly determine the next committee $\mathcal{P}^{l+1}$. There are multiple methods for selecting which servers will be in each committee, we want our evaluation to be independent of these specific selection methods.

## 6.2 Evaluation Setup

In our evaluation, the client set $N_c$ contains at most $t_c$ corrupted parties, while the server set (i.e., the committee) $N_s$ contains at most $t_s$ corrupted parties. Our evaluation setup is designed to answer the following questions:

- What is the latency of each sub-protocol?
- What is the impact of circuit shape on the latency of our protocols?
- Can $\Pi_{\mathsf{AD\text{-}MPC}}$ achieve our goals in comparison to Fluid MPC and HoneybadgerMPC?

[1] https://github.com/zkcrypto/pairing
[2] https://github.com/python/cpython

Protocol $\Pi_{\mathsf{Trans}}$

- **Input:** The shared value $[z_k^l]$, the random value $[\alpha_k^l]$ (for $k \in [w]$).
- **Output:** The sharing $[z_k^{l+1}]$ after updating the polynomial.
- **Procedure:**
  **Committee $\mathcal{P}^l$:**
  1. $P_i^l$ (for $i \in [n]$) locally computes $m_{k,i}^l = z_{k,i}^l + \alpha_{k,i}^l$.
  2. $P_i^l$ calls $\mathcal{F}_{\mathsf{Commit}}$ with $\alpha_{k,i}^l$ as input to get commitment $\pi_{\alpha_{k,i}^l}$.
  3. $P_i^l$ invokes $\mathcal{F}_{\mathsf{RBC}}$ with $m_{k,i}^l$ and $\pi_{\alpha_{k,i}^l}$ as inputs to $\mathcal{P}^{l+1}$.
  4. $P_i^l$ invokes $\mathcal{F}_{\mathsf{ACSS}}$ with $z_{k,i}^l$ as input to $\mathcal{P}^{l+1}$.

  **Committee $\mathcal{P}^{l+1}$:**
  5. Request output from $\mathcal{F}_{\mathsf{ACSS}}$. Upon receiving valid ACSS instance $(z_{(k,i),j}^l, \pi_{z_{k,i}^l})$ from $P_i^l$, $P_j^{l+1}$ invokes $\mathcal{F}_{\mathsf{Commit}}$ with $(\pi_{z_{k,i}^l}, \pi_{\alpha_{k,i}^l}, m_{k,i}^l)$ as input, executing $w$ times in parallel. If $\mathcal{F}_{\mathsf{Commit}}$ outputs all 1, $P_j^{l+1}$ puts $i$ into a set $LT_j$. Otherwise, $P_j^{l+1}$ discards the share $z_{i,j}^l$.
  6. When $|LT_j| = 2t+1$, $P_j^{l+1}$ calls $\mathsf{OEC}$ with $LT_j$ as input to find a set $GT_j$ that contains $2t+1$ valid shares that interpolate a polynomial $p_j(\cdot)$ satisfying $p_j(0) = m^l$.
  7. Define predicate $P(\cdot)$ outputs 1 only when $|GT_j| = n-t$ and $GT_j \in LT_j$. $P_j^{l+1}$ calls $\mathcal{F}_{\mathsf{MVBA}}$ with $(GT_j, LT_j, P(\cdot))$ as input.
  8. Request output form $\mathcal{F}_{\mathsf{MVBA}}$ until receiving $T$, where $|T| = n-t$. Servers in $\mathcal{P}^{l+1}$ select the first $t+1$ elements from the set $T$ to form the common subset $CS$. $P_j^{l+1}$ locally computes $z_{k,j}^{l+1} = \mathsf{Lagrange}(|CS|, \{(1, z_{(k,1),j}^l), \cdots, (|CS|, z_{(k,|CS|),j}^l)\}, j)$

Figure 7: Protocol $\Pi_{\mathsf{Trans}}$

- What is the impact of circuit size on the latency of $\Pi_{\mathsf{AD\text{-}MPC}}$, Fluid MPC and HoneybadgerMPC?

For the first question, we fix a layered arithmetic circuit and vary the committee size to test the latency of each sub-protocol, thereby identifying the bottleneck of the main-protocol $\Pi_{\mathsf{AD\text{-}MPC}}$. Specifically, we design a rectangular arithmetic circuit with depth $d$ and width $w$, where the number of multiplication gates in each layer is $\lfloor w/2 \rfloor$, and the rest are composed of linear gates such as addition. For $l \in \{1, \cdots, d-1\}$ and $k \in \{1, \cdots, w-1\}$, the input of the circuit gate $z_k^{l+1}$ comes from $z_k^l$ and $z_{k+1}^l$, and the input of $z_w^{l+1}$ comes from $z_w^l$ and $z_{w-1}^l$. We fix the circuit's inputs to $2w$, provided by clients.

For the second question, we modify the shape of the circuit to discuss its impact on our protocols. Specifically, we generate rectangular circuits and adjust their depth and width simultaneously, but keep the number of circuit gates consistent across different circuits. The rest of the circuit settings are the same as above. Under this condition, we measure the latency of main-protocol and sub-protocols under the same committee settings.

For the third question, we mainly focus on the impact of asynchronous and dynamic settings on guaranteed output delivery. We compare $\Pi_{\mathsf{AD\text{-}MPC}}$ with Fluid MPC to analyze the impact of asynchronous setting on guaranteed output delivery. Given that Fluid MPC assumes a synchronous network, in an asynchronous network, the committee $\mathcal{P}^{l+1}$ may not receive all shares from $\mathcal{P}^l$. This could cause $\mathcal{P}^{l+1}$ to become stuck during the committee handover, preventing further circuit evaluation. Therefore, we modify Fluid MPC to accommodate asynchronous networks. Specifically, in the semi-honest adversary model, we allow $\mathcal{P}^{l+1}$ to proceed with the circuit evaluation after receiving $n-t$ shares from $\mathcal{P}^l$ (Fluid MPC 1.0). Furthermore, under the malicious adversary model, we enhance it to allow $\mathcal{P}^{l+1}$ to call the MVBA protocol, ensuring the consistency of the set used for Lagrange interpolation. Each committee contains up to $t$ corrupted parties. We simulate these corrupted parties implementing additive attacks during committee handovers and observe the correctness of the protocol.

Additionally, we compare $\Pi_{\mathsf{AD\text{-}MPC}}$ with HoneybadgerMPC to discuss the impact of dynamic setting on guaranteed output delivery. To maintain consistency in other properties, we replace its non-robust offline phase with our sub-protocol $\Pi_{\mathsf{Aprep}}$ to implement the fully asynchronous HoneybadgerMPC protocol. Under the malicious adversary model, we establish a single committee to evaluate the circuit, where the adversary can control up to $t$ servers. Subsequently, we randomly select other servers to remain silent during the circuit evaluation, simulating the impact of dynamic party behavior on guaranteed output delivery. In the malicious adversary model, our main-protocol $\Pi_{\mathsf{AD\text{-}MPC}}$ runs under asynchronous and dynamic settings. we observe whether it can achieve guaranteed output delivery in these conditions. For the final question, we fix the circuit width and committee size, and test the latency of protocols ($\Pi_{\mathsf{AD\text{-}MPC}}$, Fluid MPC and HoneybadgerMPC) by changing the circuit depth.

We deploy $\Pi_{\mathsf{AD\text{-}MPC}}$ along with Fluid MPC and HoneyBadgerMPC, across a Wide Area Network (WAN) using 20 *C6.2XLARGE32* instances from different cities. To simulate multiple parties, we configure each instance to run the pro-

tocols using different ports. Each instance is equipped with a 16-Core CPU, 32GB memory, and 50GB SSD, and is configured with Debian 12.0. The bandwidth capacity of each instance is 100Mbps. We simulate twenty times and present the average for each data point.

## 6.3 Evaluation Results

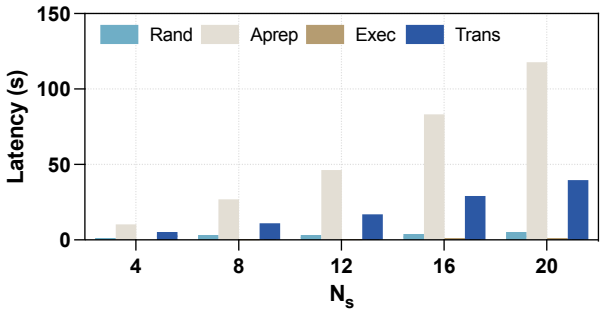


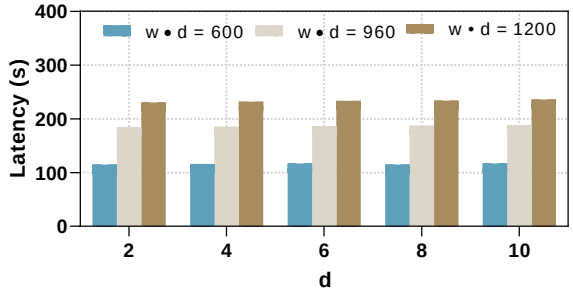

Figure 8: The latency of sub-protocols. We fix the circuit depth $d = 6$ and width $w = 100$, and change the committee size $n = 4, 8, 12, 16, 20$ to evaluate the latency of each sub-protocol. The ratio of multiplication gates to linear gates in each layer is 1:1. Exec represents the circuit evaluation.

### 6.3.1 The latency of each sub-protocol

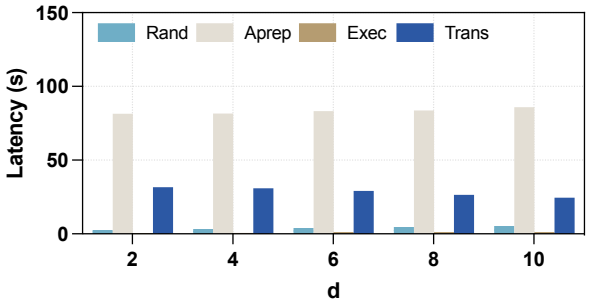
We set up a layered arithmetic circuit with depth $d = 6$ and width $w = 100$, and test the total latency of each sub-protocol under different committee sizes ($n = 4, 8, 12, 16, 20$). The ratio of multiplication gates to linear gates in each layer is 1:1. The results are shown in Figure 8. We observe that $\Pi_{\text{Aprep}}$ consistently accounts for the largest proportion of latency, which gradually increases from 62.3% to 72.3% as the committee size increases. The reason $\Pi_{\text{Aprep}}$ is so time-consuming is that each server in $\mathcal{P}^l$ independently generates Beaver triples required for the next committee $\mathcal{P}^{l+1}$, as well as an equal number of additional triples to verify the correctness of these Beaver triples. Consequently, each committee $\mathcal{P}^l$ needs to generate a total of $n \cdot w$ triples. By calling $\Pi_{\text{Aprep}}$, we can obtain correct triples while maintaining optimal resilience in asynchronous networks, without relaxing network assumptions during the preprocessing [39]. Compared to $\Pi_{\text{Aprep}}$, the latency of the circuit evaluation Exec is significantly smaller. When committee size is 20, the delay is reduced by approximately 117.18s. Therefore, our protocol will perform better in circuits with sparse multiplication gates.

### 6.3.2 The impact of circuit shape on latency

We fix the number of gates in the circuit to $w \cdot d = 600, 960, 1200$ and change the depth $d$ and the corresponding width $w$ to evaluate the latency of $\Pi_{\text{AD-MPC}}$. Figure 9(a) shows our results, with the protocol latency gradually increasing as the circuit depth increases. Compared to $d = 2$, when $d = 10$, with $w \cdot d = 600, 960$ and 1200, the protocol delays increased by approximately 2.00s, 3.99s, and 6.09s, respectively. To explain the reasons, we measure the latency of each sub-protocol when $w \cdot d = 600$, as shown in Figure 9(b). We find that the latency of $\Pi_{\text{Aprep}}$ increases with the number of circuit layers, while the latency of $\Pi_{\text{Trans}}$ decreases. Although both $\Pi_{\text{Aprep}}$ and $\Pi_{\text{Trans}}$ have a communication complexity of $O(\kappa n^2 + \kappa n^3)$, the coefficient in front of $\kappa n^3$ for $\Pi_{\text{Aprep}}$ is greater than that for $\Pi_{\text{Trans}}$. Consequently, the overall latency tends to increase as the number of circuit layers increases.




(a)




(b)

Figure 9: (a) The latency of $\Pi_{\text{AD-MPC}}$ under different circuit shapes. We set the number of gates in the circuit to 600, 960 and 1200 (i.e., $w \cdot d = 600, 960, 1200$) and then vary both the depth $d$ and width $w$ simultaneously to test the latency of $\Pi_{\text{AD-MPC}}$ under committee size $n = 16$. (b) The latency of sub-protocols for each circuit when $w \cdot d = 600$.

### 6.3.3 The impact of asynchronous and dynamic settings on guaranteed output delivery

We set the circuit width $w = 100$, depth $d = 6$, and committee size $n = 16$, allowing for up to $t = 5$ corrupted parties. We then evaluate the progress of different protocols ($\Pi_{\text{AD-MPC}}$, Fluid MPC 1.0, Fluid MPC 2.0 and HoneybadgerMPC). The 7-th layer represents the output stage where clients receive

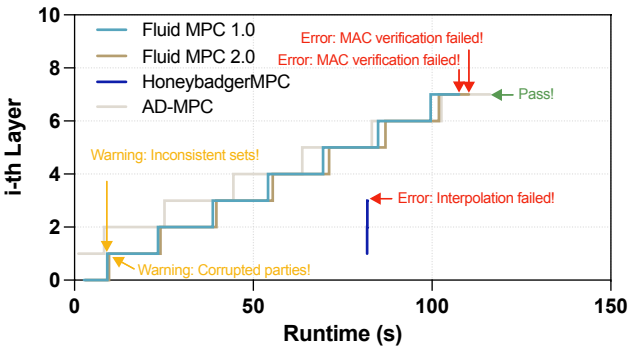



Figure 10: The progress of $\Pi_{\text{AD-MPC}}$, Fluid MPC 1.0, Fluid MPC 2.0 and HoneybadgerMPC. We fix the circuit width $w = 100$, the circuit depth $d = 6$, and the committee size $n = 16$. The (7)-th layer represents the phase where clients receive shares and attempt to reconstruct them.

and verify shares. Note that Fluid MPC adds an additional layer (the 0-th layer) to compute a MAC of each client input and incrementally compute two random linear combinations. For HoneybadgerMPC, we simulate an honest server going offline during the evaluation of the 4-th layer and observe the protocol's behavior. The results are shown in Figure 10.

While running Fluid MPC 1.0, we observe that at the 1-th layer, different servers in $\mathcal{P}^1$ produced six distinct sets (Warning: Inconsistent sets!), leading to the interpolation of six different degree-$t$ polynomials, though their constant terms remain the original secrets. By the 2-th layer, each server in $\mathcal{P}^2$ uses a set for interpolation that includes at least two polynomials from $\mathcal{P}^1$. This results in new polynomials whose constant terms are no longer the original secrets, causing incorrect outputs (Error: MAC verification failed!). In all twenty of our tests, inconsistent sets are found at the 2-th layer, leading to incorrect outputs. This indicates that Fluid MPC 1.0 cannot achieve guaranteed output delivery in an asynchronous network.

When running Fluid MPC 2.0, we find that although all servers in the committee can interpolate new polynomials from a consistent set, this set already includes additive errors from corrupted parties (Warning: Corrupted parties!), which leads to wrong outputs (Error: MAC verification failed!). This demonstrates that merely adding asynchronous primitives cannot achieve guaranteed output delivery.

For HoneybadgerMPC, the protocol terminates because when an honest server goes offline, the remaining honest servers cannot call OEC to interpolate a correct polynomial during the multiplication evaluation (Error: Interpolation failed!). This indicates that considering dynamic settings can affect protocols that were originally able to guarantee output delivery. The reason for the steeper progress of HoneybadgerMPC is that its primary overhead is incurred during the preprocessing phase, which involves calling $\Pi_{\text{Aprep}}$.

In our main-protocol $\Pi_{\text{AD-MPC}}$, although it operates in asynchronous and dynamic settings, it still achieves guaranteed output delivery, thus meeting our goals set out in Section 3.1. Moreover, $\Pi_{\text{AD-MPC}}$ is only slower by approximately 8.693s and 5.894s compared to Fluid MPC 1.0 and 2.0, respectively.

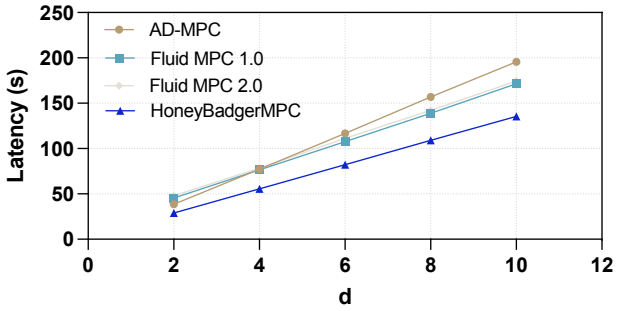



Figure 11: The latency of $\Pi_{\text{AD-MPC}}$, Fluid MPC 1.0, Fluid MPC 2.0 and HoneybadgerMPC. We fix the circuit width $w = 100$ and committee size $n = 16$, and change the circuit depth $d = 2, 4, 6, 8, 10$.

#### 6.3.4 The impact of circuit size on latency

We also test the impact of different circuit sizes on these protocols. We fix the circuit width $w = 100$, the committee size $n = 16$, and vary the circuit depth $d$ to 2, 4, 6, 8, and 10. The results are shown in Figure 11. We find that the overhead of all protocols increases linearly with increasing circuit depth, indicating that these protocols maintain stable processing efficiency across each layer. The latency for both Fluid MPC 1.0 and 2.0 is higher than that of HoneybadgerMPC, indicating that the dynamic setting incurs higher overhead compared to the asynchronous setting. This observation also explains why the slope of the curve for $\Pi_{\text{AD-MPC}}$ is steeper.

## 7 Conclusion

In this paper, we proposed $\Pi_{\text{AD-MPC}}$, a fully asynchronous dynamic MPC protocol that achieves guaranteed output delivery with optimal resilience (i.e., $n = 3t + 1$). To the best of our knowledge, $\Pi_{\text{AD-MPC}}$ is the first dynamic MPC protocol designed for asynchronous networks. We formally prove the security of $\Pi_{\text{AD-MPC}}$ within the Universal Composability (UC) framework. Furthermore, we provide a prototype implementation and evaluate its performance using geographically distributed nodes. The result shows that $\Pi_{\text{AD-MPC}}$ and Fluid MPC exhibit comparable runtimes, $\Pi_{\text{AD-MPC}}$ achieves guaranteed output delivery, whereas Fluid MPC only achieves security with abort.

## Ethics Considerations and Open Science Policy

## Research Ethics Considerations

We ensure that our research adheres to the four ethical principles outlined in the Menlo Report: "Respect for Persons," "Beneficence," "Justice," and "Respect for Law and Public Interest." The specific ethical considerations are detailed below:

- **Respect for Persons:** This research has been approved by all authors, developers, and contributors involved. Since our research focuses on the MPC protocol, which does not involve human subjects, no additional volunteers were required. All external code referenced in our experiments has been appropriately cited with links provided in the paper.
- **Beneficence:** Our research does not involve external data, thereby ensuring that there is no risk of privacy violations for individuals or organizations. We also confirm that the deployment of our protocol complies with all relevant laws.
- **Justice:** The interests of all authors, developers, and contributors involved in this research were fully considered to ensure fairness and equity.
- **Respect for Law and Public Interest:** We ensure that our research complies with all relevant laws and regulations. Additionally, we have open-sourced our code and provided detailed descriptions of our protocols in the paper, ensuring compliance, transparency, and accountability.

## Compliance with Open Science Policy

The research adheres to the principles of open science, ensuring transparency, replicability, and accessibility. We have open-sourced our code and datasets, allowing other researchers to replicate our experiments and validate our results. Furthermore, our paper includes detailed descriptions of protocols, algorithms, and experimental setups, ensuring that the processes involved are clear and understandable to others.

## References


[1] Ittai Abraham, Gilad Asharov, Shravani Patil, and Arpita Patra. Perfect asynchronous mpc with linear communication overhead. Cryptology ePrint Archive, 2024.

[2] Gilad Asharov and Yehuda Lindell. A full proof of the bgw protocol for perfectly secure multiparty computation. Journal of Cryptology, 30(1):58–151, 2017.

[3] Aritra Banerjee, Michael Clear, and Hitesh Tewari. zkhawk: Practical private smart contracts from mpc-based hawk. In 2021 3rd Conference on Blockchain Research & Applications for Innovative Networks and Services (BRAINS), pages 245–248. IEEE, 2021.

[4] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. An end-to-end system for large scale p2p mpc-as-a-service and low-bandwidth mpc for weak participants. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pages 695–712, 2018.

[5] Carsten Baum, James Hsin-yu Chiang, Bernardo David, and Tore Kasper Frederiksen. Eagle: Efficient privacy preserving smart contracts. In International Conference on Financial Cryptography and Data Security, pages 270–288. Springer, 2023.

[6] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In Theory of Cryptography Conference, pages 213–230. Springer, 2008.

[7] Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous secure computation. In Proceedings of the twenty-fifth annual ACM symposium on Theory of computing, pages 52–61, 1993.

[8] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali, pages 351–371. 2019.

[9] 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.

[10] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In Annual International Conference on the Theory and Applications of Cryptographic Techniques, pages 169–188. Springer, 2011.

[11] Fabrice Benhamouda, Craig Gentry, Sergey Gorbunov, Shai Halevi, Hugo Krawczyk, Chengyu Lin, Tal Rabin, and Leonid Reyzin. Can a public blockchain keep a secret? In Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part I 18, pages 260–290. Springer, 2020.

[12] Alexander Bienstock, Daniel Escudero, and Antigoni Polychroniadou. On linear communication complexity for (maximally) fluid mpc. Cryptology ePrint Archive, 2023.

[13] Gabriel Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In Proceedings of the third annual ACM symposium on Principles of distributed computing, pages 154–162, 1984.

[14] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. white paper, 3(37):2–1, 2014.

[15] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In Annual International Cryptology Conference, pages 524–541. Springer, 2001.

[16] Ran Canetti. Studies in secure multiparty computation and applications. Scientific Council of The Weizmann Institute of Science, 1996.

[17] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In Proceedings 42nd IEEE Symposium on Foundations of Computer Science, pages 136–145. IEEE, 2001.

[18] Ran Canetti, Yevgeniy Dodis, Rafael Pass, and Shabsi Walfish. Universally composable security with global setup. In Theory of Cryptography: 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007. Proceedings 4, pages 61–85. Springer, 2007.

[19] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. Fluid mpc: secure multiparty computation with dynamic participants. In Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part II 41, pages 94–123. Springer, 2021.

[20] Ashish Choudhury, Martin Hirt, and Arpita Patra. Asynchronous multiparty computation with linear communication complexity. In International Symposium on Distributed Computing, pages 388–402. Springer, 2013.

[21] Ashish Choudhury, Emmanuela Orsini, Arpita Patra, and Nigel P Smart. Linear overhead optimally-resilient robust mpc using preprocessing. In International Conference on Security and Cryptography for Networks, pages 147–168. Springer, 2016.

[22] Ashish Choudhury and Arpita Patra. Optimally resilient asynchronous mpc with linear communication complexity. In Proceedings of the 16th International Conference on Distributed Computing and Networking, pages 1–10, 2015.

[23] Ashish Choudhury and Arpita Patra. An efficient framework for unconditionally secure multiparty computation. IEEE Transactions on Information Theory, 63(1):428–468, 2016.

[24] Ashish Choudhury and Arpita Patra. On the communication efficiency of statistically secure asynchronous mpc with optimal resilience. Journal of Cryptology, 36(2):13, 2023.

[25] Ran Cohen. Asynchronous secure multiparty computation in constant time. In Public-Key Cryptography–PKC 2016, pages 183–207. Springer, 2016.

[26] Sandro Coretti, Juan Garay, Martin Hirt, and Vassilis Zikas. Constant-round asynchronous multi-party computation based on one-way functions. In International Conference on the Theory and Application of Cryptology and Information Security, pages 998–1021. Springer, 2016.

[27] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In Annual Cryptology Conference, pages 643–662. Springer, 2012.

[28] Sourav Das, Zhuolun Xiang, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling. In 32nd USENIX Security Symposium (USENIX Security 23), pages 5359–5376, 2023.

[29] Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In 2022 IEEE Symposium on Security and Privacy (SP), pages 2518–2534. IEEE, 2022.

[30] Giovanni Deligios, Aarushi Goel, and Chen-Da Liu-Zhang. Maximally-fluid mpc with guaranteed output delivery. Cryptology ePrint Archive, 2023.

[31] Daniel Genkin, Yuval Ishai, and Antigoni Polychroniadou. Efficient multi-party computation: from passive to active security via secure simd circuits. In Annual Cryptology Conference, pages 721–741. Springer, 2015.

[32] 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.

[33] Yossi Gilad, Rotem Hemo, Silvio Micali, Georgios Vlachos, and Nickolai Zeldovich. Algorand: Scaling byzantine agreements for cryptocurrencies. In Proceedings

of the 26th symposium on operating systems principles, pages 51–68, 2017.

[34] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game, or a completeness theorem for protocols with honest majority. In Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali, pages 307–328. 2019.

[35] Vipul Goyal, Chen-Da Liu-Zhang, and Yifan Song. Towards achieving asynchronous mpc with linear communication and optimal resilience. Cryptology ePrint Archive, 2024.

[36] Bingyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Dumbo: Faster asynchronous bft protocols. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 803–818, 2020.

[37] Yehuda Lindell. Secure multiparty computation. Communications of the ACM, 64(1):86–96, 2020.

[38] Ziyao Liu, Ivan Tjuawinata, Chaoping Xing, and Kwok-Yan Lam. Mpc-enabled privacy-preserving neural network training against malicious attack. arXiv preprint arXiv:2007.12557, 2020.

[39] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pages 887–903, 2019.

[40] Satoshi Nakamoto and A Bitcoin. A peer-to-peer electronic cash system. Bitcoin.–URL: https://bitcoin.org/bitcoin. pdf, 4(2):15, 2008.

[41] Peter Sebastian Nordholt and Meilof Veeningen. Minimising communication in honest-majority mpc by batchwise multiplication verification. In International Conference on Applied Cryptography and Network Security, pages 321–339. Springer, 2018.

[42] Arpita Patra, Ashish Choudhury, and C Pandu Rangan. Efficient asynchronous verifiable secret sharing and multiparty computation. Journal of Cryptology, 28:49–109, 2015.

[43] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Annual international cryptology conference, pages 129–140. Springer, 1991.

[44] Rahul Rachuri and Peter Scholl. Le mans: Dynamic and fluid mpc for dishonest majority. In Annual International Cryptology Conference, pages 719–749. Springer, 2022.

[45] Ryan Wails, Aaron Johnson, Daniel Starin, Arkady Yerukhimovich, and S Dov Gordon. Stormy: Statistics in tor by measuring securely. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, pages 615–632, 2019.

[46] Andrew Chi-Chih Yao. How to generate and exchange secrets. In 27th annual symposium on foundations of computer science (Sfcs 1986), pages 162–167. IEEE, 1986.

## A Randomness Extraction

We utilize the randomness extraction technique based on the hyper-invertible matrix [6]. A hyper-invertible matrix is one where every square sub-matrix is invertible. By utilizing a hyper-invertible matrix, parties can extract $n-t$ uniform random values from $n$ input secrets through local linear operations. Taking $\{[x_1],\cdots[x_n]\}$ as our $n$ input sharings, parties use the following Vandermonde Matrix and locally compute the subsequent formula to obtain $n-t$ random sharings:

$$\begin{bmatrix} [r_1] \\ \vdots \\ [r_{n-t}] \end{bmatrix} = \begin{bmatrix} 1 & \omega_1 & \cdots & \omega_1^{n-1} \\ 1 & \omega_2 & \cdots & \omega_2^{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{n-t} & \cdots & \omega_{n-t}^{n-1} \end{bmatrix} \begin{bmatrix} [s_1] \\ [s_2] \\ \vdots \\ [s_n] \end{bmatrix} \quad (1)$$

Since the matrix is hyper-invertible, if at least $n-t$ input sharings are uniformly random and independent, we can ensure that $n-t$ output sharings are also uniformly random and independent. We refer to the randomness extraction technique as RE and will employ this technique in subsequent protocols.

## B Computing Linear Functions of t-sharings

$t$-sharings are linear. Specifically, given $k$ $t$-sharings $\{[x_1],\cdots,[x_k]\}$ and a publicly known linear function $g: \mathbb{F}^k \to \mathbb{F}^l$, where $g(x_1,\cdots,x_k) = (y_1,\cdots,y_l)$, it holds that $g([x_1],\cdots,[x_k]) = ([y_1],\cdots,[y_l])$. Each party $P_i \in \mathcal{P}$ can locally compute $(y_{1,i},\cdots,y_{l,i}) = g(x_{1,i},\cdots,x_{k,i})$, where $\{x_{1,i},\cdots,x_{k,i}\}$ are shares held by $P_i$.

Our protocol employs the standard Lagrange's polynomial-evaluation function Lagrange to compute a "new" point on a polynomial based on "old" points [24]. Specifically, Lagrange takes a set $\mathcal{K}$ as input, which contains $|\mathcal{K}|$ distinct value pairs $\{(x_1, y_1),\cdots,(x_{|\mathcal{K}|}, y_{|\mathcal{K}|})\}$. Let $p(\cdot)$ be a degree-$(|\mathcal{K}|-1)$ polynomial that satisfies $p(x_i) = y_i$. For $x_j \in \mathbb{Z}_q \backslash \{x_1,\cdots,x_{|\mathcal{K}|}\}$, by calling the function Lagrange, we can obtain a corresponding $y_j$ such that $p(x_j) = y_j$ holds.

Here, there exist publicly known Lagrange's coefficients [2] $\{c_1,\cdots,c_{|\mathcal{K}|}\}$ associated with $\{x_1,\cdots,x_{|\mathcal{K}|},x_j\}$ which satisfy $y_j = c_1 \cdot x_1 + \cdots + c_{|\mathcal{K}|} \cdot x_{|\mathcal{K}|}$. We represent this computation as follows:

$$y_j = \mathsf{Lagrange}(|\mathcal{K}|, \{(x_1,y_1),\cdots,(x_{|\mathcal{K}|},y_{|\mathcal{K}|})\}, x_j) \quad (2)$$

## C Robust Interpolation

Our protocol utilizes the online error-correction (OEC) protocol [7, 16] for robust interpolation. Consider a degree-$t$ polynomial $p(\cdot)$, where $p(0) = s$. In the protocol, every party sends its share to each other. Parties then invoke the Reed-Solomon (RS) error-correction procedure on the received shares. If $t < n/3$, parties can correct the received corrupt shares and reconstruct a new degree-$t$ polynomial $p'(\cdot)$, satisfying $p'(\cdot) = p(\cdot)$ and $p'(0) = s$. We denote the resultant protocol as OEC and employ it in our protocols.

## D Some Ideal Functionalities

Figure 12 shows our ideal functionality $\mathcal{F}_{\mathsf{ACSS}}$, which captures the properties we propose. implementations of the ACSS protocol are available in [24, 28, 29], so we omit their instances for brevity.

**Ideal Functionality $\mathcal{F}_{\mathsf{ACSS}}$**

$\mathcal{F}_{\mathsf{ACSS}}$ proceeds as follows, running with parties $\mathsf{Set} = \{P_s^l, P_1^{l+1}, \cdots, P_n^{l+1}\}$ and an adversary $\mathcal{S}$, where $P_s^l$ represents the sender in committee $\mathcal{P}^l$.

- Upon receiving (dealer, ACSS, $id$, $p(\cdot)$) from $P_s^l$, proceed as follows:
  - If $p(\cdot)$ is a degree-$t$ polynomial, store $(p(0), \pi_{p(0)})$, where $\pi_{p(0)}$ is a unused identifier to represent $p(0)$. Generate a request-based delayed output $(P_s^l, \mathsf{ACSS}, id, p(i), \pi_{p(0)})$ for each $P_i^{l+1} \in \mathsf{Set}$.
  - If $p(\cdot)$ is not a degree-$t$ polynomial, generate a request-based delayed output $(P_s^l, \mathsf{ACSS}, id, \perp)$ for each $P_i^{l+1} \in \mathsf{Set}$.

Figure 12: Ideal Functionality $\mathcal{F}_{\mathsf{ACSS}}$

The ideal functionality is shown in Figure 13. Bracha [13] proposed an elegant RBC protocol, and Choudhury [24] gave a UC proof of their protocol.

**Ideal Functionality $\mathcal{F}_{\mathsf{RBC}}$**

$\mathcal{F}_{\mathsf{RBC}}$ proceeds as follows, running with parties Set and an adversary $\mathcal{S}$. Assume that $\mathcal{P}_c$ is the set of corrupted parties, where $|\mathcal{P}_c| \le t$.

- Upon receiving (sender, RBC, $id$, $m$) from $P_i$, send ($P_i$, RBC, $id$, $m$) to $\mathcal{S}$ and generate a request-based delayed output ($P_i$, RBC, $id$, $m$) for each $P_j \in \mathsf{Set} \backslash \{\mathcal{P}_c, P_i\}$.

Figure 13: Ideal Functionality $\mathcal{F}_{\mathsf{RBC}}$

Figure 14 illustrates the ideal functionality $\mathcal{F}_{\mathsf{MVBA}}$, which captures the aforementioned properties. While numerous MVBA protocol implementations already exist (e.g., [28, 29, 36]), we will omit specific details here for brevity.

**Ideal Functionality $\mathcal{F}_{\mathsf{MVBA}}$**

$\mathcal{F}_{\mathsf{MVBA}}$ proceeds as follows, running with parties Set and an adversary $\mathcal{S}$. Assume that $\mathcal{P}_c^l$ is the set of corrupted parties, where $|\mathcal{P}_c^l| \le t$.

- Upon receiving (MVBA, $id$, $LT_i$, $GT$, $P(\cdot)$) from $P_i \in \mathsf{Set}$, where $GT$ represents a public set accessible to all parties. If $P(LT_i, GT) = 1$, record $LT_i$. Otherwise, ignore the message.
- After recording $n-t$ $LT_i$ that satisfy predicate $P(\cdot)$, select a set $T$ from these valid sets and generate a request-based delayed output $T$ for all parties.

Figure 14: Ideal Functionality $\mathcal{F}_{\mathsf{MVBA}}$

In our paper, we leverage the standard commitment functionality, denoted as $\mathcal{F}_{\mathsf{Commit}}$ in Figure 15.

**Ideal Functionality $\mathcal{F}_{\mathsf{Commit}}$**

$\mathcal{F}_{\mathsf{Commit}}$ proceeds as follows, running with parties Set and an adversary $\mathcal{S}$.

- Upon receiving (Commit, $id$, $m$, $\pi_m$) from $P_i \in \mathsf{Set}$, where $\pi_m$ is a previously unused identifier, store ($id$, $m$, $\pi_m$), and send ($id$, $\pi_m$) to all parties.
- Upon receiving (Verify, $id$, $\pi_x$, $\pi_y$, $z$) from $P_i \in \mathsf{Set}$, retrieve $x$ and $y$, check whether $z = x + y$. If the check passes, output 1 to all parties. Otherwise, output 0 to all parties.

Figure 15: Ideal Functionality $\mathcal{F}_{\mathsf{Commit}}$

## E Security Analysis

**Security Model.** We follow the dynamic MPC model introduced by [12, 19, 44], based on the UC framework [17, 18]. To capture the asynchronous setting, we refer to the models in [24, 25, 35]. Specifically, to model the scenario where an adversary can control when each honest party receives the output from an ideal functionality $\mathcal{F}$, we introduce the concept of activations. When $\mathcal{F}$ has an output for some parties, the party requests $\mathcal{F}$ for output, and the adversary can instruct $\mathcal{F}$ to delay the output. The party will eventually receive the output within a polynomial number of activations. As in [24, 25, 35], we use the term $\mathcal{F}$ *sends a request-based delayed output to* $P_i$ to describe such behavior.

Our ideal functionality, $\mathcal{F}_{\mathsf{AD\text{-}MPC}}$, depicted in Figure 16, operates over a finite field $\mathbb{Z}_q$. The functionality is initialed by clients $\mathcal{P}^0$ through an **Init** input, with the current layer $i$ and committee $\mathcal{P}^i$ controlled by variable $i$; the first committee $\mathcal{P}^1$ is determined by $\mathcal{P}^0$. $\mathcal{P}^0$ provides secrets through **Input**, assigning each variable $x_i$ a public identifier $id_{x_i}$. During the execution stage, the committee evaluates addition and multiplication by inputting **Add** and **Multiply**, respectively. The current committee inputs **Next-Committee** to determine the following committee. Finally, $\mathcal{P}^0$ retrieves the result by inputting **Output**.

**Ideal Functionality $\mathcal{F}_{\text{AD-MPC}}$**

**Parameters:** Finite field $\mathbb{Z}_q$ where $q$ is prime, the total number of clients $\mathcal{P}^0$ and each committee $\mathcal{P}$ is $n = 3t+1$ where $t$ is the number of corrupted parties. Each variable $x$ used in the computation has a public identifier $id_x$.
**Init:** On input $(\mathsf{Init}, \mathcal{P})$ from every $C_j \in \mathcal{C}$, where each $P_i^0$ sends the same set $\mathcal{P}$, initialize $i = 1$, $\mathcal{P}^1 = \mathcal{P}$ as the first active committee. Generate a request-based delayed output $(\mathsf{Init}, \mathcal{P}^1)$.
**Input:** Upon receiving $(\mathsf{Input}, id_{x_j}, x_j)$ from $C_j \in \mathcal{C}$, and $(\mathsf{Input}, id_{x_j})$ from all other parties in $\mathcal{P}^0$, store the pair $(id_{x_j}, x_j)$. Generate a request-based delayed output $(\mathsf{Input}, id_{x_j})$.
**Next-Committee:** Upon receiving $(\mathsf{Next\text{-}Committee}, \mathcal{P})$ from every $P_j^i \in \mathcal{P}^i$, where each $P_j^i$ sends the same set $\mathcal{P}$, sets $i = i+1$, $\mathcal{P}^i = \mathcal{P}$. Generate a request-based delayed output $(\mathsf{Next\text{-}Committee}, \mathcal{P}^i)$.
**Add:** Upon receiving $(\mathsf{Add}, id_x, id_y, id_z)$ from every $P_j^i \in \mathcal{P}^i$, compute $z = x + y$ and store $(id_z, z)$. Generate a request-based delayed output $(\mathsf{Add}, id_x, id_y, id_z)$.
**Multiply:** Upon receiving $(\mathsf{Multiply}, id_x, id_y, id_z)$ from every $P_j^i \in \mathcal{P}^i$, compute $z = x \cdot y$ and store $(id_z, z)$. Generate a request-based delayed output $(\mathsf{Multiply}, id_x, id_y, id_z)$.
**Output:** Upon receiving $(\mathsf{Output}, id_z)$ from every $C_j \in \mathcal{C}$, where $id_z$ has been stored previously, retrieve $(id_z, z)$ and output $(\mathsf{Output}, id_z, z)$.

Figure 16: Ideal Functionality $\mathcal{F}_{\text{AD-MPC}}$

**Lemma 1.** The output of protocol $\Pi_{\mathsf{Rand}}$ is simulatable with random values.

*Proof.* In the committee $\mathcal{P}^l$, the value $z_i^l$ chosen by each honest server $P_i^l$ is not known to the adversary $\mathcal{A}$. According to the security of Shamir secret sharing, the $t$ shares of $[z_i^l]$ from each honest dealer $P_i^l$ received by corrupted servers in $\mathcal{P}^{l+1}$ are uniformly random and independent. In an asynchronous network, the completeness property of $\mathcal{F}_{\mathsf{ACSS}}$ ensures that all honest servers in $\mathcal{P}^{l+1}$ obtain ACSS instances from the same degree-$t$ polynomial, even if the dealer in $\mathcal{P}^l$ is corrupted. The termination property of $\mathcal{F}_{\mathsf{MVBA}}$ ensures that all honest servers in $\mathcal{P}^{l+1}$ can terminate, and the agreement and external validity properties of $\mathcal{F}_{\mathsf{MVBA}}$ ensure that all honest servers agree on an output $T$ from $\mathcal{F}_{\mathsf{MVBA}}$ and $|T| = n - t$. Randomness extraction $\mathsf{RE}$ can extract $n - t$ independent and uniformly random values from the set $T$. □

**Lemma 2.** The correctness of Protocol $\Pi_{\mathsf{Rec}}$ is assured. Assuming the original secret is $z$, upon calling $\Pi_{\mathsf{Rec}}$, the output $z'$ obtained by all honest parties should satisfy $z' = z$.

*Proof.* The agreement and totality properties of $\mathcal{F}_{\mathsf{RBC}}$ guarantee that all honest parties receive shares broadcast by their honest counterparts. Therefore, each honest party will receive at least $2t+1$ correct shares. The correctness property of $\mathsf{OEC}$ guarantees that from these shares, each honest party can interpolate the correct degree-$t$ polynomial, where the constant term is the original secret.

□

**Lemma 3.** The output of protocol $\Pi_{\mathsf{Aprep}}$ is simulatable with random values.

*Proof.* The proof idea in the sharing random Beaver triples phase is similar to Lemma 1. The triples $(a_{(i,g)}^l, b_{(i,g)}^l, c_{(i,g)}^l)$ and $(x_{(i,g)}^l, y_{(i,g)}^l, z_{(i,g)}^l)$ chosen by each honest server $P_i^l$ in $\mathcal{P}^l$ are not known to the adversary $\mathcal{A}$. By the security of Shamir secret sharing, any $t$ shares of $([a_{(i,g)}^l], [b_{(i,g)}^l], [c_{(i,g)}^l])$ and $([x_{(i,g)}^l], [y_{(i,g)}^l], [z_{(i,g)}^l])$ from each honest dealer $P_i^l$ received by corrupted servers in $\mathcal{P}^{l+1}$ are uniformly random and independent. The completeness of $\mathcal{F}_{\mathsf{ACSS}}$ ensures that honest servers in $\mathcal{P}^{l+1}$ only receive shares that are correctly distributed on a degree-$t$ polynomial. Lemma 1 ensures that the random value generated by $\Pi_{\mathsf{Rand}}$ are perceived as uniformly random and independent from the $\mathcal{A}$'s standpoint. Servers in committee $\mathcal{P}^{l+1}$ invoke $\mathsf{OEC}$ to recover the random value $r_i^l$. Following this step, in line with Lemma 2, the integrity of $r_i^l$ is verified. Finally, the protocol sacrifices the triple $(x_{(i,g)}^l, y_{(i,g)}^l, z_{(i,g)}^l)$ to confirm $c_{(i,g)}^l = a_{(i,g)}^l \cdot b_{(i,g)}^l$, thereby ensuring the correctness of the triple $(a_{(i,g)}^l, b_{(i,g)}^l, c_{(i,g)}^l)$.
During the phase of agreeing on a common subset, the termination property of $\mathcal{F}_{\mathsf{MVBA}}$ ensures that all honest servers in $\mathcal{P}^{l+1}$ can terminate. The agreement and external validity properties of $\mathcal{F}_{\mathsf{MVBA}}$ ensure that all honest servers agree on an output $T$ where $|T| = n - t$. Through $\mathcal{F}_{\mathsf{MVBA}}$, all honest servers in $\mathcal{P}^{l+1}$ have access to a set $T$ of valid triples.
After executing the above two steps, the set $T$ contains at most $t$ valid triples from $\mathcal{A}$ that satisfy $c_{(i,g)}^l = a_{(i,g)}^l \cdot b_{(i,g)}^l$. Therefore, the last two steps of $\Pi_{\mathsf{Aprep}}$ involve re-randomizing these valid triples to ensure that the generated triples are uniformly random and independent from the $\mathcal{A}$'s perspective. For each $g \in [c_m]$, new polynomials $U_g(\cdot)$, $V_g(\cdot)$ and $W_g(\cdot)$ are interpolated using triples in $T$, with $W_g(\cdot) = U_g(\cdot) \cdot V_g(\cdot)$. Since $U_g(\cdot)$ and $V_g(\cdot)$ are degree-$t$ polyn omials requiring $t+1$ shares to interpolate, and considering that $T$ contains at most $t$ triples from $\mathcal{A}$, $U_g(\cdot)$, $V_g(\cdot)$ and $W_g(\cdot)$ are rendered uniformly random for $\mathcal{A}$. Consequently, the secret sharing over these three polynomials also appears uniformly random and independent from $\mathcal{A}$'s perspective.

□

**Lemma 4.** Protocol $\Pi_{\mathsf{Mult}}$ is simulatable.

*Proof.* According to Lemma 3, $\Pi_{\mathsf{Aprep}}$ can be simulated using random values. Meanwhile, Lemma 2 ensures the correctness of $\gamma^l$ and $\delta^l$ outputted according to $\Pi_{\mathsf{Rec}}$, allowing for their simulation with random values as well. □

**Lemma 5.** Protocol $\Pi_{\mathsf{Trans}}$ is correct and simulatable. Assuming the secret $[z^l]$ inputted into the protocol corresponds to the degree-$t$ polynomial $p(\cdot)$, and the secret $[z^{l+1}]$ outputted corresponds to the degree-$t$ polynomial $\hat{p}(\cdot)$. $\Pi_{\mathsf{Trans}}$ ensures that $p(0) = \hat{p}(0)$, and the coefficients of $\hat{p}(\cdot)$ are uniformly random and independent from the adversary's perspective.

*Proof.* In the committee $\mathcal{P}^l$, based on Lemma 1, we can conclude that $\Pi_{\mathsf{Rand}}$ is simulatable with random values. There-

fore, the masked value $m_i^l$ does not expose the secret $z_i^l$. The agreement and totality properties of $\mathcal{F}_{\mathsf{RBC}}$ guarantee that all honest servers in $\mathcal{P}^{l+1}$ receive shares $(m_i^l, \pi_{\alpha_i^l})$ broadcast by honest server $P_i^l$. The completeness of $\mathcal{F}_{\mathsf{ACSS}}$ ensures that honest servers in $\mathcal{P}^{l+1}$ only receive shares that are correctly distributed on a degree-$t$ polynomial. $\mathcal{F}_{\mathsf{Commit}}$ ensures the relationship between the masked value $m_i^l$ and the secret $z_i^l$. The correctness property of $\mathsf{OEC}$ ensures that each honest server can interpolate a degree-$t$ polynomial $p(\cdot)$ based on $[m^l]$, satisfying $p(0) = m^l$. The termination property of $\mathcal{F}_{\mathsf{MVBA}}$ ensures that all honest servers can terminate. The agreement and external validity properties of $\mathcal{F}_{\mathsf{MVBA}}$ further ensure that all honest servers agree on an output $T$. When all honest servers apply $\mathsf{OEC}$ to the set $T$, they can reconstruct a consistent polynomial whose constant term is the original secret. The coefficients of the polynomial are uniformly random and independent.

□

**Theorem 1.** Let $\mathcal{A}$ be an R-*adaptive* adversary in $\Pi_{\mathsf{AD\text{-}MPC}}$. $\Pi_{\mathsf{AD\text{-}MPC}}$ *UC-securely* computes $\mathcal{F}_{\mathsf{AD\text{-}MPC}}$ in the presence of $\mathcal{A}$ within the $(\mathcal{F}_{ACSS}, \mathcal{F}_{\mathsf{MVBA}}, \mathcal{F}_{\mathsf{RBC}}, \mathcal{F}_{\mathsf{Commit}})$-hybrid model.

*Proof.* We construct a simulator $\mathcal{S}$ to run the adversary $\mathcal{A}$ as a subroutine, which also has access to $\mathcal{F}_{\mathsf{AD\text{-}MPC}}$. It internally emulates functionalities $\mathcal{F}_{ACSS}$, $\mathcal{F}_{\mathsf{MVBA}}$, $\mathcal{F}_{\mathsf{RBC}}$ and $\mathcal{F}_{\mathsf{Commit}}$. $\mathcal{S}$ relays all communication between the adversary $\mathcal{A}$ and the environment $\mathcal{Z}$. It tracks the current committee through input $(\mathsf{Init}, \mathcal{P})$ and $(\mathsf{Next\text{-}Committee}, \mathcal{P})$ from $\mathcal{F}_{\mathsf{AD\text{-}MPC}}$. The simulation proceeds as follows:

**Input:** Upon all (even corrupted) clients sending inputs to $\mathcal{F}_{\mathsf{AD\text{-}MPC}}$, call $\mathcal{F}_{ACSS}$ with corresponding inputs, invoke $\Pi_{\mathsf{Rand}}$ as in Lemma 1 and $\Pi_{\mathsf{Aprep}}$ as in Lemma 3.

**Next-Committee:** Upon all (even corrupted) servers sending $(\mathsf{Next\text{-}Committee}, \mathcal{P})$ to $\mathcal{F}_{\mathsf{AD\text{-}MPC}}$, run $\Pi_{\mathsf{Trans}}$ as in Lemma 5.

**Addition, Addition by constant, Multiplication by constant:** Need not be simulated as they are local operations.

**Multiplication:** Upon all (even corrupted) servers sending $(\mathsf{Mult}, id_x, id_y, id_z)$ to $\mathcal{F}_{\mathsf{AD\text{-}MPC}}$, run $\Pi_{\mathsf{Mult}}$ as in Lemma 4.

**Output:** Upon all (even corrupted) servers sending $(\mathsf{Output}, id_z)$ to $\mathcal{F}_{\mathsf{AD\text{-}MPC}}$, run $\Pi_{\mathsf{Rec}}$ as in Lemma 2.

We now argue that $\mathcal{A}$ cannot distinguish between interacting with $\mathcal{S}$ and $\mathcal{F}_{\mathsf{AD\text{-}MPC}}$. During the input stage, the secrecy property of $\mathcal{F}_{\mathsf{ACSS}}$ ensures that shares from an honest dealer are uniformly random and independent from $\mathcal{A}$'s view. Hence, this is a perfect simulation. Similarly, Lemmas 1 and 3 demonstrate that $\Pi_{\mathsf{Rand}}$ and $\Pi_{\mathsf{Aprep}}$ can be perfectly simulated using random values. In the execution stage, operations like addition, constant addition, and constant multiplication are linear and require no simulation. For multiplication evaluation, Beaver triples generated by $\Pi_{\mathsf{Aprep}}$ are indistinguishable from random by Lemma 3. Additionally, Lemma 4 establishes the perfect simulatability of $\Pi_{\mathsf{Mult}}$. The correctness and simulatability of committee handovers are guaranteed by Lemma 5. Finally, Lemma 2 guarantees the reconstructed secret's correctness during the output stage, implying the accuracy of the final output.

□