AD-MPC: Asynchronous Dynamic MPC with Guaranteed Output Delivery

Wenxuan Yu
Shandong University
Qingdao, China
haitengseat@gmail.com

Minghui Xu*
Shandong University
Qingdao, China
mhxu@sdu.edu.cn

Bing Wu
Shandong University
Qingdao, China
bwu12340@gmail.com

Sisi Duan
Tsinghua University
Beijing, China
duansisi@mail.tsinghua.edu.cn

Xiuzhen Cheng
Shandong University
Qingdao, China
xzcheng@sdu.edu.cn

Abstract

MPC-as-a-Service (MPCaaS) systems enable clients to outsource privacy-preserving computations to distributed servers, offering flexibility by adapting and configuring MPC protocols to meet diverse security requirements. However, traditional MPC protocols rely on a fixed set of servers for the entire computation process, limiting scalability. Dynamic MPC (DMPC) addresses this limitation by permitting participants to join or leave during the computation. Nevertheless, existing DMPC protocols assume synchronous networks, which can lead to failures under unbounded network delays. In this paper, we present AD-MPC, the first asynchronous dynamic MPC protocol. Our protocol ensures guaranteed output delivery under optimal resilience ($n = 3t + 1$). To achieve this, we introduce two critical components: an asynchronous dynamic preprocessing protocol that facilitates the on-demand generation of Beaver triples for secure multiplication, and an asynchronous transfer protocol that maintains consistency during party hand-offs. These components collectively ensure computation correctness and transfer consistency across participants. We implement AD-MPC and evaluate its performance across up to 20 geographically distributed nodes. Experimental results demonstrate that the protocol not only offers strong security guarantees in dynamic and asynchronous network environments but also achieves performance comparable to state-of-the-art DMPC protocols.

CCS Concepts

• **Security and privacy → Cryptography**.

Keywords

Secure multi-party computation; Asynchrony

*Corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
CCS '25, Taipei, Taiwan, China
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2025/06
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Wenxuan Yu, Minghui Xu, Bing Wu, Sisi Duan, and Xiuzhen Cheng. 2025. AD-MPC: Asynchronous Dynamic MPC with Guaranteed Output Delivery. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS '25).* ACM, New York, NY, USA, 21 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

MPC-as-a-Service (MPCaaS) systems [6] provide secure multiparty computation capabilities as a service, enabling clients to outsource privacy-preserving computations to distributed servers. These systems can dynamically configure and adapt MPC protocols to meet various security requirements, offering solutions for applications such as privacy-preserving machine learning [39], anonymous voting systems [6], and private smart contracts [5, 7]. However, traditional MPC protocols typically rely on a fixed set of servers that must remain committed throughout the entire circuit evaluation. This constraint prevents the dynamic addition or removal of servers, limiting the scalability of MPCaaS in handling complex circuits.

Recent research has introduced dynamic MPC (DMPC) [10, 18, 29, 43] to address the challenge of dynamic participation. DMPC divides an arithmetic circuit into multiple epochs, with each epoch executed by a distinct set of servers, namely a committee. A critical requirement for dynamic MPCaaS systems is ensuring Guaranteed Output Delivery (GOD) for clients. If an adversary introduces errors during the circuit evaluation, it may cause the computation to abort, resulting in unavailable service. Moreover, these systems must ensure GOD in asynchronous networks, where message delays can be arbitrary, such as in cases where MPCaaS servers experience Internet disruptions. Despite this, no Asynchronous Dynamic MPC (ADMPC)[1] protocols with the GOD property have been proposed to date. This gap motivates the development of an ADMPC protocol.

1.1 A Framework for Constructing ADMPC

Although no ADMPC protocol currently exists, we first outline a general framework to illustrate the challenges in adapting DMPC from synchronous to asynchronous environments. DMPC typically partitions an arithmetic circuit into multiple epochs, each comprising a computation phase and a hand-off phase. Transitioning

[1]ADMPC denotes a class of protocols, and the AD-MPC presented in this paper represents a specific instantiation of ADMPC.

DMPC to an asynchronous setting requires replacing the synchronous communication procedures with asynchronous ones in both phases.

Computation Phase. During the computation phase, the active committee evaluates the assigned segment of the arithmetic circuit using secret-shared inputs. Linear operations can be evaluated locally without interaction due to the linearity of secret sharing, whereas the evaluation of multiplication gates requires interactive communication among committee members. Asynchronous MPC (AMPC) protocols [4, 19, 21, 22, 35, 36, 41, 45] provide a solution for asynchronous multiplication through an offline/online method. Specifically, Beaver triples are prepared offline and later utilized in online multiplication evaluations. In our ADMPC framework, the active committee proactively generates Beaver triples offline for the subsequent committee to use in robust online multiplication, ensuring compatibility with dynamic settings.

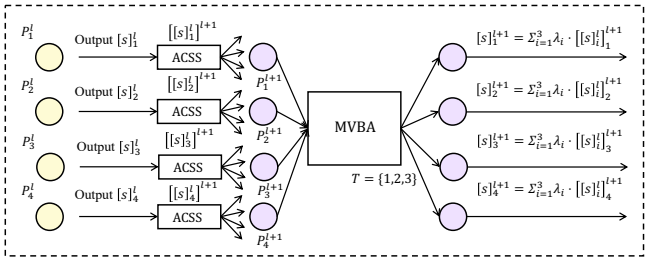
Hand-off Phase. The hand-off phase comprises two core tasks: securely transferring the secret-shared outputs to the next committee and preparing necessary Beaver triples for future multiplication gates. Unlike synchronous DMPC protocols that rely on synchronous broadcasts, asynchronous primitives must be employed in ADMPC. Specifically, each server in the active committee employs Asynchronous Complete Secret Sharing (ACSS) [27, 46] to re-share its secret-shared outputs. Subsequently, the next committee utilizes Multi-valued Validated Byzantine Agreement (MVBA) [1, 14] to achieve consensus on a common subset of valid re-shared values. This ensures all honest servers in the next committee reconstruct consistent new polynomials, as illustrated in Fig. 1(a).

Simultaneously, the current committee must prepare Beaver triples asynchronously for the next committee. We summarize two prevalent methods in existing asynchronous MPC literature, depicted in Fig. 1(b):

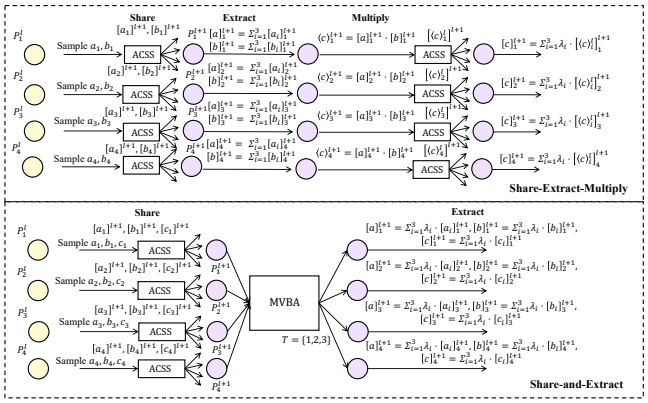
- *Share-Extract-Multiply* [4, 19, 36, 41, 45]: Each party in the current committee samples random pairs of values and distributes secret shares of these pairs to the next committee. Servers of the incoming committee then extract local shares of these random values from received shares. Each server locally multiplies its extracted shares, resulting in a degree-$2t$ polynomial. Finally, a degree-reduction step transforms this into a valid degree-t sharing, thus forming a secure Beaver triple.
- *Share-and-Extract* [21, 22, 35]: Alternatively, the current committee directly samples candidate Beaver triples and distributes them with the incoming committee. Subsequently, the next committee runs an extraction procedure on these candidates, securely extracting uniformly random and valid Beaver triples.

1.2 Challenges in Constructing ADMPC

While the framework described above outlines how to realize asynchronous computation and hand-off, two significant challenges remain when building ADMPC protocols in practice. First, asynchronous triple generation suffers from high communication complexity. Second, the framework cannot directly achieve GOD. We elaborate on these challenges below.



(a) Secret-shared Outputs Transfer Procedure



(b) Asynchronous Beaver Triple Generation Methods

Figure 1: Core procedures in the hand-off phase.

High Communication Overhead for Triple Generation. Asynchronous triple generation typically follows one of two paradigms: share-extract-multiply or share-and-extract, each with inherent limitations:

- *Share-Extract-Multiply.* GS23 [36] presents an approach well-suited to dynamic settings by extending BGW to asynchronous environments via Pedersen's polynomial commitments [42]. When instantiating ACSS from Pedersen's polynomial commitments, however, these commitments incur a communication overhead of $O(\kappa n^3)$ per Beaver triple (where κ is the security parameter).
- *Share-and-Extract.* CP23 [22] demonstrates a strong candidate for dynamic settings in the share-and-extract method, offering statistical security and optimal resilience. However, CP23 does not explicitly instantiate several critical subprotocols, particularly the random share generation functionality $\mathcal{F}_{\mathsf{Rand}}$. This gap introduces security risks in dynamic environments: corrupted servers in the incoming committee might prematurely reconstruct random values and collude with corrupted servers from the current committee. They can then produce invalid Beaver triples that erroneously pass verification. Consequently, the correctness of generated triples can be compromised. Additionally, this method suffers from a communication overhead of $O(\kappa n^4)$ per Beaver triple, which hinders practical deployment.

Maintaining Transfer Consistency for GOD. Achieving GOD necessitates the correct transfer of secret-shared outputs between committees. Beyond ensuring computational correctness within an epoch, the hand-off protocol must maintain *Transfer Consistency*. This property requires that the secret value s underlying the shares $[s]^l$ held by committee $\mathcal{P}^l$ is faithfully rerandomized by the next committee, $\mathcal{P}^{l+1}$. Specifically, when $\mathcal{P}^{l+1}$ interpolates its new sharing $[s]^{l+1}$ from the shares received during hand-off, the resulting polynomial's constant term must be the original secret s.

However, the described hand-off framework, relying on ACSS for share distribution, fails to guarantee this property. While ACSS ensures that the shares distributed by a dealer (a server in $\mathcal{P}^l$) constitute points on some valid degree-t polynomial, it lacks a mechanism to verify that this polynomial encodes the correct, original secret s held by the dealer. Consequently, a corrupted dealer in $\mathcal{P}^l$ could distribute valid-looking shares corresponding to an entirely different secret s'. If these incorrect shares are subsequently agreed upon by $\mathcal{P}^{l+1}$ (e.g., by forming the common subset determined via MVBA), the committee will incorrectly interpolate the polynomial $[s']^{l+1}$, leading to a violation of transfer consistency ($s' \neq s$).

1.3 Our Contributions

We introduce an efficient ADMPC protocol, termed AD-MPC. Our protocol achieves guaranteed output delivery under optimal resilience, specifically with committees of size $n = 3t + 1$. To address the challenges outlined above, we propose two countermeasures: (1) an asynchronous dynamic preprocessing protocol with quasi-quadratic communication complexity during the computation phase, and (2) an asynchronous transfer protocol that ensures transfer consistency during the hand-off phase. The contributions of this work are summarized as follows.

- **Asynchronous Dynamic Preprocessing with Quasi- quadratic Communication.** Building on the share-and-extract method from CP23 [22], we reduce the overhead for generating Beaver triples in an asynchronous dynamic setting from $O(\kappa n^4)$ to $O(\kappa n^2 \log n)$ per triple. Specifically, we introduce a batch random share generation protocol, BatchRand, which uses hyper-invertible matrices and the batch ACSS protocol (hbACSS) [46] to achieve $O(\kappa n \log n)$ communication for each random share. Building on BatchRand and additional batch reconstruction and multiplication sub-protocols, we develop ADprep, an asynchronous dynamic preprocessing protocol. ADprep prevents premature reconstruction of random values, thereby ensuring that corrupted parties cannot falsely pass invalid triples. Furthermore, by relying on the robustness of Shamir secret sharing in the online phase, our protocol guarantees correct multiplication throughout the computation phase.
- **Asynchronous Dynamic Transfer with Consistency.** We propose ADtrans, an asynchronous transfer protocol that ensures consistency during the hand-off phase. At its core, ADtrans introduces the *mask-and-reconstruct* method, in which each server in the current committee masks its secret-shared outputs using randomness and provides commitments that bind these masked shares to the original shares. The incoming committee verifies these commitments and then reconstructs the masked values, guaranteeing that the newly formed sharing encodes the same secret as before. To efficiently generate the required randomness, we introduce BatchBundle, a lightweight variant of BatchRand with the same asymptotic communication complexity. Moreover, we extend the batch polynomial commitment scheme (hbPolyCommit) [46] with a "transfer evaluation" interface to generate and distribute these commitments. Overall, ADtrans achieves a quasi-quadratic communication complexity.
- **Implementation of ADMPC with GOD.** We develop a prototype of AD-MPC and implement core components from Fluid MPC [18] to enable direct performance comparisons. To the best of our knowledge, this is the first implementation of an asynchronous dynamic MPC protocol. Our experimental evaluations demonstrate practical performance across a range of network conditions, showing that AD-MPC not only matches the efficiency of Fluid MPC but also provides stronger security guarantees. The practical implementation is open-sourced at https://anonymous.4open.science/r/Anonymous-Authors-ADMPC.

Table 1: Comparisons of DMPC protocols and basic approaches

DMPC	Network	$t <$	GOD	Comm./Gate[1]	Imple.[2]
Fluid MPC [18]	sync.	$n/3$	✗	$O(\kappa n^2)$	✓
Le Mans [43]	sync.	n	✗	$O(\kappa n^2)$	✗
DDG+23[3] [29]	sync.	$n/3$	✓	$O(\kappa n^2)$	✗
BEP23 [10]	sync.	n	✗	$O(\kappa n)$	✗
BEP24 [11]	sync.	$n/3$	✗	$O(\kappa n)$	✗
DKLN25 [30]	sync.	$n/2$	✓	$O(\kappa n^2)$	✗
ADMPC (instantiated via Section 1.1 with GS23 or CP23)					
SEM[4] [36]	async.	$n/3$	✗	$O(\kappa n^3)$	✗
SE[5] [22]	async.	$n/3$	✗	$O(\kappa n^4)$	✗
This work	async.	$n/3$	✓	$O(\kappa n^2 \log n)$	✓

[1] "Comm./Gate" denotes the communication complexity per multiplication gate.
[2] "Imple." indicates whether the protocol has been validated through an actual implementation.
[3] DDG+23 [29] achieves perfect security at $t < n/3$ and computational security at $t < n/2$ while achieving guaranteed output delivery.
[4] SEM refers to the share-extract-multiply, instantiated using GS23 [36].
[5] SE refers to the share-and-extract, instantiated using CP23 [22].

2 Related Work

In this section, we first review existing DMPC protocols, including both synchronous designs and YOSO-style approaches. We then discuss two prevalent Beaver triple generation methods from prior AMPC research. Table 1 summarizes representative DMPC protocols alongside two ADMPC instantiations.

2.1 DMPC

The concept of dynamic MPC (DMPC) was pioneered by Fluid MPC [18], which introduced the notion of maximal fluidity, meaning that a party can exit the computation immediately after sending and receiving messages in a single round. Subsequent DMPC protocols preserve this maximal fluidity while achieving other properties, such as security against a dishonest majority [10, 43] or linear communication complexity [10, 11]; however, they all provide only Security with Abort (SwA), enabling honest parties to detect misbehavior but not guaranteeing delivery of the output. DDG+23 [29] takes a further step by attaining GOD under maximal fluidity, offering one protocol with perfect security for $t < n/3$ and another with computational security for $t < n/2$. In addition, DKLN25 [30] demonstrates that GOD can also be achieved with statistical security when $t < n/2$. Nonetheless, all of these protocols operate under synchronous network assumptions and thus do not address the challenges arising from asynchrony. Note that in an asynchronous environment, achieving consensus among committee members requires multiple rounds of communication, so we do not pursue maximal fluidity in this paper.

Another line of work, known as the YOSO (You Only Speak Once) MPC [3, 9, 15, 16, 24, 25, 31, 34, 38, 40], also seeks maximal fluidity by hiding each committee's identity until after a single broadcast in both synchronous and asynchronous networks. Compared to DMPC, YOSO MPC adopts a different threat model and introduces additional costs for anonymous setup and membership verification. Due to these fundamental differences, we do not aim to directly compare with YOSO MPC in this work.

2.2 AMPC

To realize an ADMPC protocol, we leverage secure multiplication techniques from AMPC. In particular, AMPC protocols typically generate Beaver triples using one of two methods: share-extract-multiply or share-and-extract. Table 1 highlights existing DMPC protocols and shows two straightforward instantiations of our framework (from Section 1.1) that use GS23 [36] (for share-extract-multiply) and CP23 [22] (for share-and-extract). Below, we briefly review the key results in each method.

Share-Extract-Multiply. CHP13 [19] achieves linear communication complexity per triple, but only for adversary thresholds $t < n/4$. CP15 [20] relies on threshold decryption of somewhat homomorphic encryption for each multiplication in the online phase, which introduces significant efficiency challenges. HoneyBadgerMPC [41] is the first implemented AMPC system; however, its offline phase is not robust. GS23 [36] attains optimal resilience ($t < n/3$) by employing Pedersen's polynomial commitments [42], thus offering a robust triple generation, though at the cost of incurring a communication overhead of $O(\kappa n^3)$ per triple. Dumbo-MPC [45] further reduces communication complexity using KZG commitments, but it requires a trusted setup. Finally, BJK+24 [4] achieves linear communication complexity under optimal resilience; however, it incurs a circuit-independent overhead of $O(\kappa n^5)$, and generating each triple requires hundreds of verifiable secret sharings, which together render its concrete efficiency impractical.

Share-and-Extract. CP16 [21] implements the share-and-extract method with linear communication complexity per triple, albeit only for adversary thresholds $t < n/4$. CP23 [22] raises the adversarial threshold to the optimal $t < n/3$, but its offline phase cost increases to $O(\kappa n^4)$ per triple. Furthermore, GLS24 [35] claims linear communication complexity under the optimal threshold yet suffers from an exorbitant $O(\kappa n^{14})$ circuit-independent overhead and, similar to BJK+24, exhibits impractical concrete efficiency, with each triple requiring thousands of verifiable secret sharings when instantiated with state-of-the-art cryptographic primitives.

3 Model and Preliminaries

In this section, we formally introduce the model of ADMPC by clearly specifying its participants, network assumption, dynamism, and threat model. We also articulate the formal design goals necessary to realize ADMPC protocols. Next, we describe the cryptographic primitives and asynchronous building blocks that our protocol leverages.

3.1 Model and Design Goals

Network Assumption. We denote by $\mathcal{S} = \{P_1, P_2, \dots\}$ a finite set containing all participating servers. We assume the existence of a public-key infrastructure (PKI), where each server $P_i \in \mathcal{S}$ holds a public-private key pair (pk_i, sk_i), and all public keys pk_i are publicly known. The network is asynchronous, featuring authenticated point-to-point channels among all parties. Under this asynchronous model, there are no timing constraints, and messages may experience arbitrary delays; however, all messages sent between honest parties must eventually be delivered [8].

Dynamism. We consider a dynamic MPCaaS system operating across three consecutive stages: input, execution, and output. During the input stage, clients submit their private inputs and computational tasks to an initial committee $\mathcal{P}^1 \subseteq \mathcal{S}$. The execution stage involves evaluating the submitted tasks, represented as a layered arithmetic circuit C with depth d, partitioned into epochs $\{C^l\}_{l\in[d]}$ [18]. For each epoch $l \in [d]$, the active committee $\mathcal{P}^l \subset \mathcal{S}$ first evaluates the epoch C^l in the computation phase, and subsequently transfers the resulting secret-shared outputs to the next committee $\mathcal{P}^{l+1}$ during the hand-off phase. Concurrently, committee $\mathcal{P}^l$ refreshes the threshold cryptosystem key material required to reconfigure the common coin used for asynchronous consensus by committee $\mathcal{P}^{l+1}$. Such reconfiguration can be realized by employing recent distributed key reconfiguration techniques [32]. Finally, during the output stage, the last committee $\mathcal{P}^d$ delivers the final computation results to the clients. We do not design a specific committee election mechanism, but instead assume the presence of an external asynchronous election protocol (e.g., Asynchronous Algorand [2]). We impose no constraints on committee membership overlap; thus, servers may participate in multiple epochs, provided they refresh their key pairs appropriately [47].

Threat Model. We adopt the R-adaptive adversary model [18], wherein a probabilistic polynomial-time (PPT) adversary $\mathcal{A}$ may adaptively corrupt up to $t < n/3$ servers at the beginning of each epoch C^l with committee $\mathcal{P}^l$. Upon corrupting a server, $\mathcal{A}$ can retroactively access its entire historical state and send messages on its behalf. If a server P_i^l that participated in a prior epoch C^m ($m < l$) is corrupted in C^l, such corruption is only permitted if it

does not cause the number of corrupted servers in C^m to exceed the threshold t. Moreover, the adversary may corrupt up to c out of k clients that submit inputs during the input stage. Without loss of generality, we assume $n = k$ and $t = c$.

Note that we do not consider the mobile corruption model. In our setting, the computation and hand-off in each epoch, corresponding to a single circuit layer, finish within a short time window, leaving insufficient time for the adversary to move between servers. In contrast, the mobile corruption model assumes that each epoch may last for an extended period, for example, one epoch per day as in [13], which does not match our setting.

Design Goals. We aim to construct an asynchronous dynamic MPC protocol that achieves guaranteed output delivery under optimal resilience, against a mobile adversary. Formally, we define our goal as follows:

Definition 3.1 (Asynchronous Dynamic MPC with Guaranteed Output Delivery). An asynchronous dynamic MPC protocol for a layered arithmetic circuit with depth d and committees of size $n = 3t + 1$, achieves guaranteed output delivery if, under the presence of a R-adaptive PPT adversary $\mathcal{A}$ corrupting at most t parties per epoch, it satisfies the following properties:
Privacy: For each epoch $l \in [d]$, the adversary $\mathcal{A}$ gains no additional information beyond what is disclosed by corrupted servers and clients.
Guaranteed Output Delivery (Robustness): For each epoch $l \in [d]$, all honest servers in $\mathcal{P}^l$ eventually output the evaluation of epoch C^l on at least $n - t$ distinct inputs received from $\mathcal{P}^{l-1}$. This property is further refined into two sub-properties:

- *Computation Correctness: During the computation phase, the protocol guarantees that any arithmetic operation (linear and non-linear) performed on shared secrets results in correct outputs, detectable and correctable even in the presence of up to t corrupted servers.*
- *Transfer Consistency: During the hand-off phase, suppose committee $\mathcal{P}^l$ transfers a secret sharing $[s]^l$ to committee $\mathcal{P}^{l+1}$. The protocol ensures that honest servers in committee $\mathcal{P}^{l+1}$ reconstruct a new sharing $[s]^{l+1}$ of the same secret s.*

3.2 Preliminaries

Polynomial Commitment. Polynomial commitment forms the foundation of batch asynchronous complete secret sharing (BACSS). We utilize the state-of-the-art batch polynomial commitment scheme, hbPolyCommit [46], which supports efficient batch operations through the following algorithms:

- $\mathsf{Setup}(1^\kappa, t)$: Generate a prime-order cyclic group $\mathbb{G}$ of order p with random generators $g, h, g_0, \ldots, g_t \in \mathbb{G}$. Output public parameters $SP = (\mathbb{G}, g, h, g_0, \ldots, g_t)$.
- $\mathsf{PolyCom}(SP, \phi(\cdot))$: Given degree-$t$ polynomial $\phi(x) = \sum_{i=0}^{t} s_i x^i$, select random values $\rho \xleftarrow{\$} \mathbb{Z}_p$, and compute polynomial commitment $C = \prod_{i=0}^{t} g_i^{s_i} \cdot h^\rho$. Output commitment C.
- $\mathsf{ProveEval}(SP, i, C, \phi(i))$: Output a compact proof π_i for the evaluation of polynomial at index i.
- $\mathsf{VerifyEval}(SP, i, C, \phi(i), \pi_i)$: Verify the correctness of evaluations using the provided commitment and proof. Output 1 if the verification succeeds, otherwise output 0.

Batch ACSS. Building upon hbPolyCommit, batch ACSS (hbACSS) [46] securely shares B secrets simultaneously among n parties through two main phases:

- $\mathsf{Share}(s_1, \ldots, s_B)$: The dealer samples random degree-t polynomials $\{\phi_j(x)\}_{j\in[B]}$ satisfying $\phi_j(0) = s_j$. Each party P_i receives a set $\{(C_j, [s_j]_i, \pi_{j,i})\}_{j\in[B]}$, consisting of shares $[s_j]_i = \phi_j(i)$, corresponding polynomial commitments C_j, and evaluation proofs $\pi_{j,i}$. Parties perform batch verification of their received shares using $\pi_{j,i}$ and C_j.
- $\mathsf{Rec}(\{(C_j, [s_j]_i, \pi_{j,i})\}_{j\in[B]})$: Parties reconstruct the original secrets by performing interpolation on verified shares.

hbACSS guarantees the following properties under asynchronous conditions:

- *Correctness:* If the dealer is honest, for $k \in [B]$, each honest party P_i will eventually receive a share $\phi_k(i)$ (i.e., $[s_k]_i$) during the sharing phase.
- *Secrecy:* If the dealer is honest and no honest parties have begun the reconstruction phase, then a PPT adversary corrupting up to t parties gains no information about $\phi_k(\cdot)$ for any $k \in [B]$, except with negligible probability.
- *Completeness:* If any honest party completes the sharing phase, then for $k \in [B]$, there exists a degree-t polynomial $\phi_k(\cdot)$ over $\mathbb{Z}_p[x]$ such that $\phi_k(0) = s'_k$, and all honest parties P_i will eventually receive their shares $[s'_k]_i = \phi_k(i)$. Additionally, when the dealer is honest, $s'_k = s_k$.

Leveraging the batch technique, hbACSS achieves a communication complexity of $O(\kappa n \log n \cdot B)$ when distributing B secrets.

Concurrent Composability of hbACSS. Dumbo-MPC [45] identifies a concurrency vulnerability within hbACSS, demonstrating that a malicious dealer could reuse transcripts from honest dealers to compromise secrecy in concurrent executions. To resolve this issue, Dumbo-MPC proposes a secure patching mechanism. We adopt these techniques to ensure hbACSS supports secure and efficient concurrent executions without affecting its asymptotic communication complexity.

MVBA. A Multi-valued Validated Byzantine Agreement (MVBA) protocol [14] 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 [23, 37]:

- *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.

We employ the MVBA protocol from [26, 28] as a building block. Their protocols do not rely on threshold signatures and have a communication complexity of $O(\kappa n^3)$.

4 Asynchronous Dynamic Preprocessing

In this section, we begin by presenting our batch random share generation protocol, BatchRand (Algorithm 1), specifically tailored for asynchronous dynamic settings. BatchRand leverages BACSS and the randomness extraction technique to achieve subquadratic communication cost. Next, we introduce two additional batch subprotocols, BatchRec and BatchMul, which respectively provide robust reconstruction and batch multiplication within the same asynchronous setting. Building on these three primitives, we propose ADprep (Algorithm 2), our asynchronous dynamic preprocessing protocol that allows the current committee $\mathcal{P}^l$ to robustly generate Beaver triples for the next committee $\mathcal{P}^{l+1}$ under optimal resilience.

4.1 Batch Random Share Generation

While many asynchronous MPC protocols [41, 45] propose random share generation for static participant sets, no existing solution seamlessly adapts to the dynamic setting. Algorithm 1 describes our batch random share generation protocol, BatchRand, specifically designed for the asynchronous dynamic environment. BatchRand combines BACSS [45, 46] with the randomness extraction technique based on hyper-invertible (Vandermonde) matrices [26], thereby reducing communication overhead while providing rigorous security proofs. Concretely, BatchRand operates in three main phases:

(1) *Distributing random shares (lines 101–102).* Each server P_i^l in $\mathcal{P}^l$ first uniformly samples B random secrets and then distributes these B secrets to committee $\mathcal{P}^{l+1}$ using BACSS.

(2) *Gathering and validating BACSS instances (lines 201–208).* Each server P_j^{l+1} in $\mathcal{P}^{l+1}$ activates the corresponding BACSS instances for every potential dealer P_i^l. Upon receiving a batch of shares $\{([e_{ik}]_j^{l+1}, C_{ik}, \pi_{ik,j})\}_{k\in[B]}$ from P_i^l, P_j^{l+1} verifies the evaluation proof $\{\pi_{ik,j}\}_{k\in[B]}$ and the polynomial commitments $\{C_{ik}\}_{k\in[B]}$ in batch. Once P_j^{l+1} has collected $2t+1$ valid BACSS instances (i.e., from $2t+1$ different dealers), it invokes an MVBA instance. MVBA ensures that all honest servers in $\mathcal{P}^{l+1}$ agree on a common subset T of size $2t+1$ containing valid dealers.

(3) *Extracting random shares (lines 209–212).* After reaching consensus on the common subset T, each server P_j^{l+1} extracts $(t+1)\cdot B$ uniformly random shares from the $2t+1$ valid BACSS instances. Concretely, for each $k \in [B]$, P_j^{l+1} computes

$$[r_{1k}, \cdots, r_{(t+1)k}]_j^{l+1} \leftarrow M \cdot \left[[e_{1k}, \cdots, e_{(2t+1)k}]_j^{l+1}\right]^\top$$

where M is a $(t+1)\times(2t+1)$ Vandermonde matrix over $\mathbb{Z}_p$. This linear transformation guarantees that even if up to t dealers in T are corrupted, the resulting $(t+1)$-out-of-$(2t+1)$ extraction still yields uniformly random and independent shares. The final output $\texttt{rand_shares} = \{[r_{pk}]^{l+1}\}_{p\in[t+1],k\in[B]}$ is then returned to all honest servers in $\mathcal{P}^{l+1}$.

Security of BatchRand**.** The security properties of BatchRand, including *termination*, *consistency*, *randomness*, and *secrecy*, are formally stated and proven in Appendix A.

Complexity of BatchRand**.** We execute $\Theta(n)$ instances of BACSS in parallel to share B secrets each. A single BACSS instance for B secrets costs $O(\kappa\, n \log n \cdot B)$, leading to a total of $O(\kappa\, n^2 \log n \cdot B)$ for all dealers. Once $2t+1$ valid instances are collected, we invoke a single MVBA protocol with communication cost $O(\kappa\, n^3)$. Thus, to generate $B\cdot(t+1)$ random shares, the total communication cost is $O(\kappa\, n^2 \log n \cdot B + \kappa\, n^3)$. When $B = \Omega(n)$, the amortized cost per random share is $O(\kappa\, n \log n)$.

Algorithm 1 Batch random share generation for $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$

Inputs: $\mathcal{P}^l$ with batch size B and other system parameters
Outputs: $\mathcal{P}^{l+1}$ obtains $\texttt{rand_shares}=\{[r_{pk}]^{l+1}\}_{p\in[t+1],k\in[B]}$

Server P_i^l (in committee $\mathcal{P}^l$):
// 1. Distribute random shares via BACSS
101: sample B random secrets $(e_{i1}, \cdots, e_{iB}) \xleftarrow{\$} \mathbb{Z}_p$
102: invoke BACSS_share$_i$ as dealer to share $(e_{i1}, \cdots, e_{iB})$

Server P_j^{l+1} (in committee $\mathcal{P}^{l+1}$):
// 2. Agree on $n-t$ valid and completed dealers
201: $T_j \leftarrow \emptyset$, $\texttt{shares}_j \leftarrow \emptyset$, $\texttt{rand_shares} \leftarrow \emptyset$
202: activate BACSS_share$_i$ instance for every $i \in [n]$
203: **upon** receiving valid $\{([e_{ik}]_j^{l+1}, C_{ik}, \pi_{ik,j})\}_{k\in[B]}$ from BACSS_share$_i$
// Validity verified via VerifyEval
204: $T_j \leftarrow T_j \cup \{i\}$
205: $\texttt{shares}_j \leftarrow \texttt{shares}_j \cup \{[e_{ik}]_j^{l+1}\}_{k\in[B]}$
206: **if** $|T_j| = 2t+1$
207: call MVBA with snapshot T_j as input and T_j as state
// MVBA's predicate checks if $T_j \supseteq$ input
208: **wait** for MVBA output T, w.l.o.g, let $T = \{P_1, \cdots, P_{2t+1}\}$
// 3. Extract uniformly random and independent shares
209: **for** $k \in [B]$
210: $[r_{1k}, \cdots, r_{(t+1)k}]_j^{l+1} \leftarrow M \cdot \left[[e_{1k}, \cdots, e_{(2t+1)k}]_j^{l+1}\right]^\top$
211: $\texttt{rand_shares} \leftarrow \texttt{rand_shares} \cup \{[r_{pk}]_j^{l+1}\}_{p\in[t+1]}$
212: **return** rand_shares

4.2 Batch Reconstruction and Multiplication

We remark that both batch reconstruction protocol BatchRec and batch multiplication protocol BatchMul follow standard approaches established in [21]. Briefly, BatchRec reconstructs $t+1$ secrets by treating them as coefficients of a degree-t polynomial and leveraging the inherent robustness of Shamir secret sharing (with $n = 3t+1$). Similarly, BatchMul implements standard multiplication using Beaver triples and employs BatchRec as a sub-protocol to batch reconstruct intermediate terms. We refer the reader to Appendix B for the formal definitions and security guarantees.

Complexity of BatchRec **and** BatchMul**.** Reconstructing $t+1$ secrets via BatchRec costs $O(\kappa\, n^2)$ of communication in the asynchronous setting. Evaluating $t+1$ multiplication gates with BatchMul (via pre-generated Beaver triples) also incurs $O(\kappa\, n^2)$ of communication overall.

4.3 Asynchronous Dynamic Preprocessing

Building on the preceding protocols, we now introduce ADprep (Algorithm 2), our asynchronous dynamic preprocessing protocol that

enables committee $\mathcal{P}^l$ to robustly generate Beaver triples for the next committee $\mathcal{P}^{l+1}$. In particular, ADprep mitigates vulnerabilities that arise when random shares are prematurely reconstructed. This risk stems from the lack of an asynchronous dynamic random share generation protocol, which may otherwise allow the next committee to collude with the current one and compromise the correctness of Beaver triples. By integrating our BatchRand, BatchRec, and BatchMul protocols, ADprep proceeds in four main phases:

(1) *Distributing Beaver triples and random shares (lines 101–105).* Each server P_i^l in $\mathcal{P}^l$ samples two sets of random triples $\{(a_{ik}, b_{ik}, c_{ik}), (x_{ik}, y_{ik}, z_{ik})\}_{k\in[M]}$, where $c_{ik} = a_{ik} \cdot b_{ik}$ and $z_{ik} = x_{ik} \cdot y_{ik}$. The first set $\{(a_{ik}, b_{ik}, c_{ik})\}$ serves as the "*main*" triple, while the second set $\{(x_{ik}, y_{ik}, z_{ik})\}$ acts as the "*sacrificed*" triple to verify correct multiplication. Each P_i^l shares these via BACSS with $\mathcal{P}^{l+1}$. Simultaneously, all servers in $\mathcal{P}^l$ run BatchRand with input $N = n/(t+1)$ to generate random shares $\{[r_k]^{l+1}\}_{k\in[n]}$ needed by $\mathcal{P}^{l+1}$ for the "sacrifice trick".

(2) *Verifying multiplication via the "sacrifice trick" (lines 201–217).* Upon receiving valid BACSS shares for both the main and sacrificed triples from a dealer P_i^l, each server P_j^{l+1} waits for the random shares $\texttt{rand_shares} = \{[r_k]_j^{l+1}\}_{k\in[n]}$ produced by BatchRand. Only after these random shares are available does P_j^{l+1} broadcast its own copy $[r_i]_j^{l+1}$, allowing all honest servers in $\mathcal{P}^{l+1}$ to robustly reconstruct the dealer's random value r_i.
This careful ordering is critical: by ensuring that P_j^{l+1} broadcasts its random share only after it has received a valid BACSS instance from P_i^l, the protocol requires the dealer to activate its BACSS instance before any random share reconstruction can occur. The completeness property of BACSS guarantees that once P_i^l has initiated BACSS, it cannot delay or withhold its shares until after the random value is reconstructed. Consequently, any attempt by a malicious dealer to postpone its BACSS messages in order to allow premature reconstruction of random values and facilitate collusion is effectively prevented.
With the correctly reconstructed random value r_i in hand, every server then employs the standard "sacrifice trick" to verify that $c_{ik} = a_{ik} \cdot b_{ik}$ for each triple. This step ensures that any incorrect multiplication introduced by a dishonest dealer is promptly detected.

(3) *Agreeing on a common subset (lines 218–220).* Once a server P_j^{l+1} identifies $2t+1$ valid triple sets from $\mathcal{P}^l$, it initiates the MVBA protocol to agree on a common subset T of size $2t+1$. All honest servers in $\mathcal{P}^{l+1}$ will terminate MVBA with the same set T of valid dealers.

(4) *Extracting random Beaver triples (lines 221–233).* Because up to t of the $2t+1$ valid dealers may be corrupted, $\mathcal{P}^{l+1}$ must extract a uniformly random triple from these partially corrupted triples. For each triple $k \in [M]$, P_j^{l+1} collects the first $t+1$ valid triples from T and interprets them as the degree-t polynomials $f_k(\cdot)$ and $g_k(\cdot)$ where $[f_k(p)]_j^{l+1} = [a_{pk}]_j^{l+1}$ and $[g_k(p)]_j^{l+1} = [b_{pk}]_j^{l+1}$ for $p \in [t+1]$. For the remaining t points $p \in [t+2, 2t+1]$, use Lagrange interpolation to derive $[f_k(p)]_j^{l+1}$ and $[g_k(p)]_j^{l+1}$. This ensures the entire polynomial domains $\{1, \ldots, 2t+1\}$ are available for both $f_k(\cdot)$ and $g_k(\cdot)$.
The next step constructs a degree-$2t$ polynomial $h_k(\cdot) = f_k(\cdot) \cdot g_k(\cdot)$. For the first $t+1$ points $p \in [t+1]$, shares of $[f_k(p) \cdot g_k(p)]_j^{l+1}$ come directly from $[c_{pk}]_j^{l+1}$. For the remaining t points $p \in [t+2, 2t+1]$, the committee invokes BatchMul (consuming the other t valid triples in T) to securely compute $[f_k(p) \cdot g_k(p)]_j^{l+1}$. Hence, each server P_j^{l+1} obtains $[h_k(p)]_j^{l+1} = [f_k(p) \cdot g_k(p)]_j^{l+1}$ for all $p \in [2t+1]$. Finally, by evaluating $(f_k(\beta), g_k(\beta), h_k(\beta))$ at some distinct point $\beta \notin \{0, \ldots, 2t+1\}$, each server P_j^{l+1} obtains a uniformly random Beaver triple $\left([f_k(\beta)]_j^{l+1}, [g_k(\beta)]_j^{l+1}, [h_k(\beta)]_j^{l+1}\right)$ for the kth multiplication gate.

Security of ADprep**.** The security properties of ADprep, including *termination*, *validity*, and *secrecy*, are formally stated and proven in Appendix C.

Complexity of ADprep**.** We begin by executing $\Theta(n)$ instances of BACSS in parallel to share $6M$ secrets, resulting in a total cost of $O(\kappa n^2 \log n \cdot M)$. Simultaneously, the committee $\mathcal{P}^l$ calls BatchRand to provide n random shares $\{[r_i]^{l+1}\}$, adding $O(\kappa n^2 \log n)$ to the overall budget. Next, the next committee $\mathcal{P}^{l+1}$ reconstructs $\{r_i\}$ at a cost of $O(\kappa n^3)$. During the "sacrifice trick", servers run two rounds of BatchRec per dealer, each costing $O(\kappa n \cdot M)$; across $\Theta(n)$ dealers, this leads to $O(\kappa n^2 \cdot M)$ total. After identifying $2t+1$ valid dealers, $\mathcal{P}^{l+1}$ invokes a single MVBA protocol at $O(\kappa n^3)$. Finally, extracting the final Beaver triples via BatchMul costs $O(\kappa n^2 \cdot M)$. Summing these components, the total communication for generating M random Beaver triples is $O\left(\kappa n^2 \log n \cdot M + \kappa n^3\right)$.

5 Asynchronous Dyanmic Transfer

In this section, we introduce ADtrans, an asynchronous dynamic transfer protocol built upon a *mask-and-reconstruct* method. Each server masks its original share by using fresh randomness, producing a masked share that can be publicly verified without revealing sensitive information. We extend the hbPolyCommit scheme by introducing a lightweight transfer evaluation interface, enabling the receiving committee to efficiently verify these masked shares. Once a sufficient number of masked shares have been verified, the receiving committee reconstructs the masked secret and securely interpolates it into a fresh sharing, thereby ensuring transfer consistency across epochs.

5.1 Enhanced hbPolyCommit for Transfer Consistency

We begin by enhancing the hbPolyCommit scheme [46] with a transfer evaluation interface, enabling servers to prove consistency between masked shares and the committed polynomial without revealing sensitive information.

Modified Polynomial Commitment. Originally, hbPolyCommit commits to a polynomial $\phi(x) = \sum_{i=0}^{t} s_i x^i$ by sampling randomness $\rho \xleftarrow{\$} \mathbb{Z}_p$ and computing a single commitment $C = \prod_{i=0}^{t} g_i^{s_i} \cdot h^{\rho}$. To

Algorithm 2 Asynchronous dynamic preprocessing for $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$

Inputs: $\mathcal{P}^l$ holds the number of required multiplication gates M for $\mathcal{P}^{l+1}$, and other system parameters
Outputs: $\mathcal{P}^{l+1}$ obtains

$$\texttt{triples} = \left\{([f_k(\beta)]^{l+1}, [g_k(\beta)]^{l+1}, [h_k(\beta)]^{l+1})\right\}_{k\in[M]}$$

Server P_i^l (in committee $\mathcal{P}^l$):

// 1. Distribute Beaver triples and random shares

sample secrets $\{(a_{ik}, b_{ik}), (x_{ik}, y_{ik})\}_{k\in[M]} \xleftarrow{\$} \mathbb{Z}_p$
for $k \in [M]$
  $c_{ik} \leftarrow a_{ik} \cdot b_{ik}, z_{ik} \leftarrow x_{ik} \cdot y_{ik}$
invoke BACSS_share$_i$ to share $\{(a_{ik}, b_{ik}, c_{ik}), (x_{ik}, y_{ik}, z_{ik})\}_{k\in[M]}$
activate BatchRand instance with input $B = n/(t+1)$

Server P_j^{l+1} (in committee $\mathcal{P}^{l+1}$):

// 2. Validate multiplication via the "sacrifice trick"

$T_j \leftarrow \emptyset$, $\texttt{mul_shares}_j \leftarrow \emptyset$, $\texttt{triples} \leftarrow \emptyset$
activate BACSS_share$_i$ instance for every $i \in [n]$
upon receiving valid shares $\{([a_{ik}]_j^{l+1},[b_{ik}]_j^{l+1},[c_{ik}]_j^{l+1}), ([x_{ik}]_j^{l+1},[y_{ik}]_j^{l+1},[z_{ik}]_j^{l+1})\}_{k\in[M]}$ from BACSS_share$_i$
// Validity verified via VerifyEval; omitting commitments and proofs.
  wait to receive $\texttt{rand_shares} = ([r_1]_j^{l+1}, \cdots, [r_n]_j^{l+1})$
  broadcast $[r_i]_j^{l+1}$ and **wait** until r_i is reconstructed
  for $k \in [M]$
    $[\rho_{ik}]_j^{l+1} \leftarrow r_i \cdot [a_{ik}]_j^{l+1} - [x_{ik}]_j^{l+1}$
    $[\sigma_{ik}]_j^{l+1} \leftarrow [b_{ik}]_j^{l+1} - [y_{ik}]_j^{l+1}$
  activate $\lceil \frac{2M}{t+1} \rceil$ instances of BatchRec on $\{[\rho_{ik}]_j^{l+1}, [\sigma_{ik}]_j^{l+1}\}_{k\in[M]}$
  wait until they return $\{\rho_{ik}, \sigma_{ik}\}_{k\in[M]}$
  for $k \in [M]$
    $[\tau_{ik}]_j^{l+1} \leftarrow r_i \cdot [c_{ik}]_j^{l+1} - [z_{ik}]_j^{l+1} - \sigma_{ik} \cdot [x_{ik}]_j^{l+1} - \rho_{ik} \cdot [y_{ik}]_j^{l+1} - \rho_{ik} \cdot \sigma_{ik}$
  activate $\lceil \frac{M}{t+1} \rceil$ instances of BatchRec on $\{[\tau_{ik}]_j^{l+1}\}_{k\in[M]}$
  wait until they return $\{\tau_{ik}\}_{k\in[M]}$
  if $\tau_{ik} = 0$ for $\forall\, k \in [M]$
    $T_j \leftarrow T_j \cup \{i\}$
    $\texttt{mul_shares}_j \leftarrow \texttt{mul_shares}_j \cup \{([a_{ik}]_j^{l+1},[b_{ik}]_j^{l+1}, [c_{ik}]_j^{l+1})\}_{k\in[M]}$

// 3. Agree on $n-t$ valid dealers

  if $|T_j| = 2t+1$
    call MVBA using a snapshot of T_j as input and T_j as state
      // MVBA's predicate checks if $T_j \supseteq$ input
wait for MVBA output T, w.l.o.g, let $T = \{P_1, \cdots, P_{2t+1}\}$

// 4. Extract random Beaver triples

for $k \in [M]$
  for $p \in [t+1]$
    $[f_k(p)]_j^{l+1} \leftarrow [a_{pk}]_j^{l+1}$, $[g_k(p)]_j^{l+1} \leftarrow [b_{pk}]_j^{l+1}$
    // Set up degree-t polynomials $f_k(\cdot)$ and $g_k(\cdot)$
  for $p \in [t+2, 2t+1]$
    $[f_k(p)]_j^{l+1} \leftarrow \Sigma_{i=1}^{t+1} \lambda_i [f_k(i)]_j^{l+1}$
    $[g_k(p)]_j^{l+1} \leftarrow \Sigma_{i=1}^{t+1} \lambda_i [g_k(i)]_j^{l+1}$
    // λ_i is the Lagrange coeff. for eval at p
activate $\lceil \frac{2Mt}{t+1} \rceil$ instances of BatchMul on $([f_k(p)]_j^{l+1},[g_k(p)]_j^{l+1})$ and triples $([a_{pk}]_j^{l+1},[b_{pk}]_j^{l+1},[c_{pk}]_j^{l+1})$ for $k \in [M]$ and $p \in [t+2, 2t+1]$
wait until they return $\{[f_k(p) \cdot g_k(p)]_j^{l+1}\}_{k\in[M],p\in[t+2,2t+1]}$
for $k \in [M]$
  for $p \in [2t+1]$
    $[h_k(p)]_j^{l+1} \leftarrow [f_k(p) \cdot g_k(p)]_j^{l+1}$
    // Construct degree-$2t$ polynomial $h_k(\cdot) = f_k(\cdot) \cdot g_k(\cdot)$
  $\texttt{triples} \leftarrow \texttt{triples} \cup \{([f_k(\beta)]_j^{l+1}, [g_k(\beta)]_j^{l+1}, [h_k(\beta)]_j^{l+1})\}$
  // Predefined distinct point $\beta \in \mathbb{Z}_p \setminus \{0, \cdots, 2t+1\}$
return $\texttt{triples}$

facilitate consistency checks specifically for the constant term, we introduce an additional random value $\hat{s} \xleftarrow{\$} \mathbb{Z}_p$ and decompose the commitment into two distinct parts as follows:

$$\text{PolyCom}(SP, \phi(\cdot), \hat{s}) : \mathbf{C} = (g_0^{s_0} \cdot h^{\hat{s}}, \prod_{i=1}^{t} g_i^{s_i} \cdot h^{\rho}).$$

Multiplying these two components naturally recovers the original hbPolyCommit commitment, i.e., $C = \mathbf{C}[0] \cdot \mathbf{C}[1]$. Hence, all other components of hbPolyCommit, including ProveEval and VerifyEval, remain unchanged.

Transfer Evaluation Interface. Given a polynomial $\phi(\cdot)$, its associated commitment $\mathbf{C}$, and the previously sampled random value $\hat{s}$, we furhter sample fresh randomness $r, \hat{r} \xleftarrow{\$} \mathbb{Z}_p$. Our enhanced interface introduces two new algorithms for securely verifying consistency between a masked share and the committed polynomial:

- ProveConsis$(SP, \phi(\cdot), \hat{s}, r, \hat{r})$: Compute $m = \phi(0) + r$, $\hat{m} = \hat{s} + \hat{r}$ and $\omega = g_0^r h^{\hat{r}}$, output $(m, \hat{m}, \omega)$.
- VerifyConsis$(SP, \mathbf{C}, m, \hat{m}, \omega)$: If $g_0^m \cdot h^{\hat{m}} = \mathbf{C}[0] \cdot \omega$, output 1, otherwise output 0.

Security Properties. Our modification maintains the essential guarantees of *correctness*, *polynomial binding*, *evaluation binding*, *transfer binding* and *hiding*, based on the discrete logarithm assumption in $\mathbb{G}$. We refer the reader to Appendix D for formal definitions and proofs.

5.2 Asynchronous Dyanmic Transfer

Before detailing the protocol, we first outline how auxiliary randomness is generated. During the hand-off phase in epoch C^{l-1}, the committee $\mathcal{P}^{l-1}$ invokes BatchRand (Algorithm 1) to obtain a batch of random shares $\{[\hat{s}_k]^l\}_{k\in[B]}$. Subsequently, they execute the

lightweight variant BatchBundle (see Appendix E), which outputs, for each k, a random bundle $([r_k]^l, [\hat{r}_k]^l, W_k)$, where $W_k = g_0^{r_k} h^{\hat{r}_k}$. These random shares and bundles are then consumed by our protocol in epoch C^l.

Algorithm 3 describes our asynchronous dynamic transfer protocol, ADtrans, which securely transfers secret shares from committee $\mathcal{P}^l$ to $\mathcal{P}^{l+1}$. The protocol consists of four primary phases:

(1) *Masking and re-sharing secret shares (lines 101–104).* Each server $P_i^l \in \mathcal{P}^l$ holds B secret shares $\{[s_k]_i^l\}_{k\in[B]}$, along with corresponding random shares and bunldes. For each secret, the server computes a polynomial commitment, masks its share using the ProveConsis algorithm. The server then re-shares the original secret shares using the enhanced BACSS protocol, broadcasting the masked shares along with their corresponding commitments to committee $\mathcal{P}^{l+1}$.

(2) *Validating the relation between masked shares and secret shares (lines 201–206).* Each server $P_j^{l+1} \in \mathcal{P}^{l+1}$ activates instances of BACSS_share$_i$ and verifies each masked share using the VerifyConsis algorithm. Specifically, servers check whether $g_0^{[m_k]_i^l} \cdot h^{[\hat{m}_k]_i^l} = g_0^{[s_k]_i^l} h^{[\hat{s}_k]_i^l} \cdot g_0^{[r_k]_i^l} h^{[\hat{r}_k]_i^l}$. Successfully verified instances are stored locally. Note that this step only validates consistency between the committed polynomial and the masked share; it does not immediately prove that the constant term itself is the original secret from $\mathcal{P}^l$.

(3) *Achieving consensus on $n-t$ valid masked shares (lines 207–209).* Upon collecting at least $2t+1$ valid BACSS instances, every P_j^{l+1} selects a subset $T_j' \subseteq T_j$ with $|T_j'| = 2t+1$ such that the commitments W_k are identical across all dealers, and the masked shares support robust reconstruction. P_j^{l+1} then submits T_j' to MVBA. MVBA guarantees that all honest servers agree on a common subset T, thus guaranteeing these two conditions across the committee.

(4) *Interpolating new secret shares (lines 210–214).* After MVBA outputs a common dealer set T of size $2t+1$, all servers identify a common subset $T' \subseteq T$ with $|T'| = t+1$ whose aggregated commitments $\omega_k = \prod_{i\in T'} \omega_{ki}^{\lambda_i}$ satisfy $\omega_k = W_k$ for all $k \in [B]$. Using the verified original shares from this subset, servers in $\mathcal{P}^{l+1}$ perform Lagrange interpolation to derive a new sharing $[s_k]^{l+1}$, whose constant term matches the original secret s_k.

Security of ADtrans**.** The security properties of ADtrans, including *termination*, *transfer consistency*, and *secrecy*, are formally stated and proven in Appendix F.

Complexity of ADtrans**.** ADtrans re-shares B secrets via BACSS, incurring $O(\kappa n^2 \log n \cdot B)$ of communication. After collecting at least $2t+1$ valid re-shares, one MVBA call is invoked at $O(\kappa n^3)$. Hence, the total communication cost to transfer B secrets is $O(\kappa n^2 \log n \cdot B + \kappa n^3)$.

Algorithm 3 Asynchronous dynamic transfer for $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$

Inputs: $\mathcal{P}^l$ holds the batch size B, secret shares $\{[s_k]^l\}_{k\in[B]}$, random shares $\{[\hat{s}_k]^l\}_{k\in[B]}$, random bundles $\{([r_k]^l, [\hat{r}_k]^l, W_k)\}_{k\in[B]}$ and other system parameters

Outputs: $\mathcal{P}^{l+1}$ obtains new_shares $= ([s_1]^{l+1} \cdots, [s_B]^{l+1})$

Server P_i^l (in committee $\mathcal{P}^l$):

// **1. Mask and re-share secret shares**

101: **for** $k \in [B]$

102: $\quad \mathrm{C}_{ki} \leftarrow \mathrm{PolyCom}(SP, \phi_{ki}(\cdot), [\hat{s}_k]_i^l)$, s.t. $\phi_{ki}(0) = [s_k]_i^l$

103: $\quad ([m_k]_i^l, [\hat{m}_k]_i^l, \omega_{ki}) \leftarrow \mathrm{ProveConsis}(SP, \phi_{ki}(\cdot), [\hat{s}_k]_i^l, [r_k]_i^l, [\hat{r}_k]_i^l)$

104: invoke BACSS_share$_i$ as dealer to re-share $\{[s_k]_i^l\}_{k\in[B]}$ using commitments $\{\mathrm{C}_{ki}\}_{k\in[B]}$ with broadcasting $\{[m_k]_i^l, [\hat{m}_k]_i^l, \omega_{ki}, W_k\}_{k\in[B]}$

Server P_j^{l+1} (in committee $\mathcal{P}^{l+1}$):

// **2. Validate the relation among shares**

201: $T_j \leftarrow \emptyset$, old_shares$_j \leftarrow \emptyset$, new_shares $\leftarrow \emptyset$

202: activate BACSS_share$_i$ instance for every $i \in [n]$

203: **upon** receiving valid $\{[[s_k]_i^l]_j^{l+1}, \mathrm{C}_{ki}, [m_k]_i^l, [\hat{m}_k]_i^l, \omega_{ki}, W_k\}_{k\in[B]}$ from BACSS_share$_i$

// Validity verified via VerifyEval; omitting evaluation proofs

204: $\quad$ **if** $\mathrm{VerifyConsis}(SP, \mathrm{C}_{ki}, [m_k]_i^l, [\hat{m}_k]_i^l, \omega_{ki}) = 1, \forall\, k \in [B]$

205: $\quad\quad$ old_shares$_j \leftarrow$ old_shares$_j \cup \{[[s_k]_i^l]_j^{l+1}\}_{k\in[B]}$

206: $\quad\quad T_j \leftarrow T_j \cup \{i\}$

// **3. Agree on $n-t$ valid masked shares**

207: $\quad\quad$ **if** $|T_j| \geq 2t+1$ and $\exists\, T_j' \subseteq T_j$ with $|T_j'| = 2t+1$, s.t. $m_k \leftarrow \sum_{i\in T_j'} \lambda_i [m_k]_i^l$, $\hat{m}_k \leftarrow \sum_{i\in T_j'} \lambda_i [\hat{m}_k]_i^l$ and W_k in T_j' match for $\forall\, k \in [B]$

208: $\quad\quad\quad$ call MVBA using T_j' as input and T_j as state

// MVBA's predicate checks if input satisfies Line 207

209: **wait** for MVBA output T, w.l.o.g, let $T = \{P_1, \cdots, P_{2t+1}\}$

// **4. Interpolate new secret shares**

210: select a common subset $T' \subseteq T$ with $|T'| = t+1$ such that $W_k = \prod_{i\in T'} (\omega_{ki})^{\lambda_i}$ for $\forall\, k \in [B]$

211: **for** $k \in [B]$

212: $\quad [s_k]_j^{l+1} \leftarrow \Sigma_{i\in T'} \lambda_i [[s_k]_i^l]_j^{l+1}$

// λ_i is the Lagrange coeff. for eval at 0

213: $\quad$ new_shares $\leftarrow$ new_shares $\cup \{[s_k]_j^{l+1}\}_{k\in[B]}$

214: **return** new_shares

6 Asynchronous Dynamic MPC

We now present our asynchronous dynamic MPC protocol, denoted AD-MPC, which achieves guaranteed output delivery (GOD) under optimal resilience. Fig. 2 illustrates the high-level workflow of the protocol.

The protocol proceeds epoch-by-epoch over a layered arithmetic circuit C of depth d, as defined in Section 3.1. Each epoch C^l is handled by a dedicated committee $\mathcal{P}^l$, and has width w^l, consisting of up to $m^l \leq w^l$ multiplication gates. The execution of AD-MPC follows three distinct stages:

- **Input Stage:** Clients distribute their private inputs to the first committee $\mathcal{P}^1$ using BACSS, and jointly generate both random shares and random bundles via BatchRand and BatchBundle. In addition, clients invoke ADprep to prepare

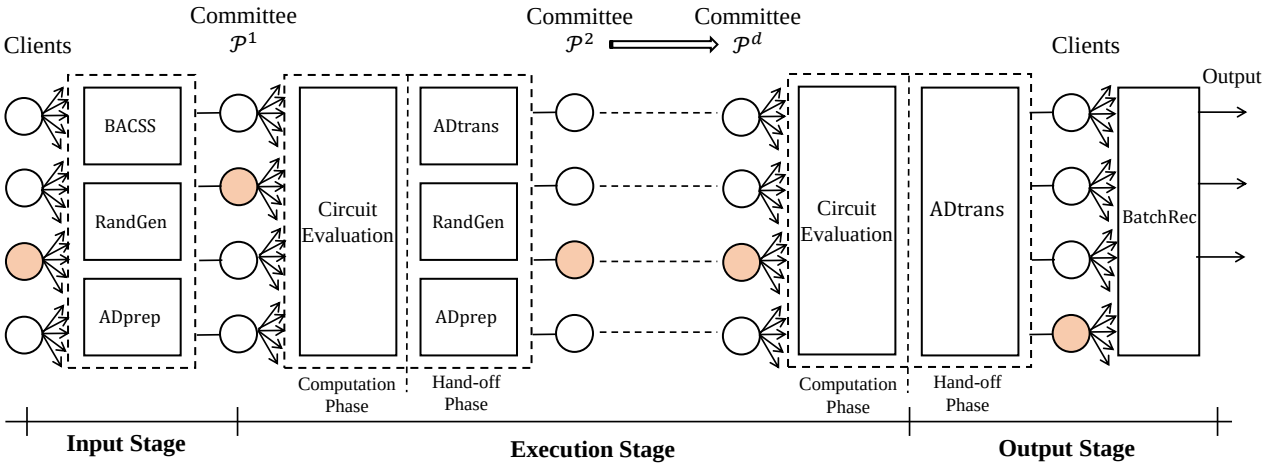


Figure 2: Workflow of AD-MPC**. For clarity, only the phases of committee $\mathcal{P}^1$ and $\mathcal{P}^d$ are explicitly shown; intermediate committees $\mathcal{P}^2, \ldots, \mathcal{P}^{d-1}$ are omitted as they follow the same procedure as $\mathcal{P}^1$.** RandGen **denotes the combined use of** BatchRand **and** BatchBundle **to generate shared randomness. Orange nodes represent corrupted parties.**

the Beaver triples required for securely evaluating the multiplication gates in epoch C^1.

- **Execution Stage:** During each epoch $l \in [d]$, committee $\mathcal{P}^l$ executes the corresponding layer C^l in two sequential phases:
  Computation Phase: Servers in $\mathcal{P}^l$ evaluate linear gates locally. For each multiplication gate, they invoke BatchMul using the Beaver triples prepared by committee $\mathcal{P}^{l-1}$, thereby securely completing the multiplication operations for epoch C^l.
  Hand-off Phase: Upon completing the computation (for all epochs except the final one, i.e., $l < d$), the committee securely transfers intermediate outputs to the next committee $\mathcal{P}^{l+1}$ via ADtrans. Simultaneously, $\mathcal{P}^l$ executes BatchRand and BatchBundle to refresh randomness, and invokes ADprep to generate the Beaver triples required for the next epoch.
- **Output Stage:** After the final committee $\mathcal{P}^d$ completes its computations, it securely transfers the resulting outputs back to the clients via ADtrans. Clients then call BatchRec to robustly reconstruct the final circuit outputs.

Security of AD-MPC**.** The security properties of AD-MPC, including *privacy* and *guaranteed output delivery* are formally stated and proven in Appendix G.

Complexity of AD-MPC**.** Consider an arithmetic circuit divided into D epochs, each of width W and containing up to $M \leq W$ multiplication gates. The per-epoch complexity is dominated by two sub-protocols: ADprep and ADtrans. Evaluating M multiplication gates via BatchMul incurs $O(\kappa\, n \cdot M)$ communication, while generating Beaver triples through ADprep requires $O(\kappa\, n^2 \log n \cdot M + \kappa\, n^3)$. Transferring W intermediate shares with ADtrans costs $O(\kappa\, n^2 \log n \cdot W + \kappa\, n^3)$. Randomness generation via BatchRand and BatchBundle incurs a lower cost of $O(\kappa\, n \log n \cdot W)$, which is asymptotically negligible.

Thus, the dominant communication complexity per epoch is $O(\kappa\, n^2 \log n \cdot W + \kappa\, n^3)$. Aggregating over all D epochs, the total communication complexity becomes $O\big((\kappa\, n^2 \log n \cdot W + \kappa\, n^3) \cdot D\big)$.

7 Implementation and Evaluation

In this section, we present our prototype implementation of AD-MPC and evaluate its performance. We analyze the latency of sub-protocols, explore the impact of circuit characteristics, and compare our protocol against existing solutions to demonstrate its effectiveness under asynchronous and dynamic conditions.

7.1 Implementation

We have implemented a prototype of AD-MPC consisting of over 5,000 lines of code. Our implementation aligns with the asynchronous dynamic model presented in Section 3.1. We utilize `bls12381`[2] for elliptic curve operations and `asyncio`[3] for asynchronous execution. Our prototype supports secure computations over $\mathbb{Z}_p$ for arbitrary layered circuits. Additionally, we have re-implemented Fluid MPC (originally closed-source) for a direct performance comparison.

7.2 Evaluation Setup

Each committee comprises $n = 3t + 1$ servers, tolerating up to t corruptions. The client set includes $k = n$ clients, of which up to $c = t$ may also be corrupted. Our evaluation addresses four key questions:

- What is the latency of each sub-protocol?
- How does circuit shape influence protocol latency?
- Can AD-MPC achieve our design goals compared to Fluid MPC and HoneybadgerMPC?
- How does circuit size affect the latency of AD-MPC, Fluid MPC, and HoneybadgerMPC?

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

For sub-protocol latency evaluation, we use rectangular arithmetic circuits with depth d and width w, maintaining a 1:1 ratio of multiplication to linear gates per layer. The input for each gate is structured sequentially, with client inputs fixed at $2w$. To analyze the impact of circuit shape, we vary depth and width simultaneously while holding the total number of gates constant.

To compare AD-MPC with Fluid MPC in asynchronous settings, we adapt Fluid MPC in two variants. In the semi-honest model, we allow $\mathcal{P}^{l+1}$ to proceed with circuit evaluation once it receives $n-t$ shares from $\mathcal{P}^{l}$ (Fluid MPC 1.0). In the malicious model, we enhance the protocol by allowing $\mathcal{P}^{l+1}$ to invoke the MVBA protocol to ensure consistency of the subset used for Lagrange interpolation (Fluid MPC 2.0). For comparison with HoneyBadgerMPC under dynamic conditions, we replace its non-robust offline phase with our ADprep protocol and simulate dynamism by randomly taking honest servers offline. Finally, we fix the circuit width and committee size, and test the latency of protocols (AD-MPC, Fluid MPC and HoneybadgerMPC) by changing the circuit depth.

We deploy our experiments on 20 geographically distributed WAN-based *C6.2XLARGE32* instances (16-Core CPU, 32GB RAM, 50GB SSD, 500Mbps bandwidth) with Debian 12.0. Each configuration is executed twenty times, and we report averaged results.

7.3 Evaluation Results

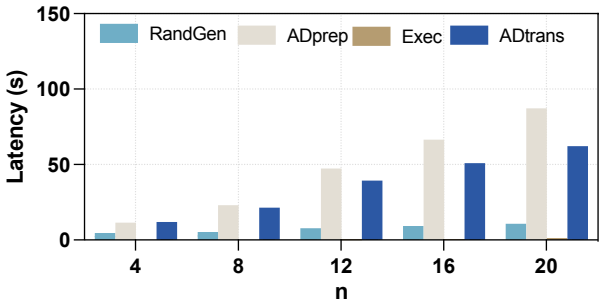


Figure 3: The latency of sub-protocols. We fix the circuit depth $d = 6$ and width $w = 100$, and change the committee size $n = 4, 8, 12, 16, 20$ to evaluate the latency of each sub-protocol. The ratio of multiplication gates to linear gates in each layer is 1:1. Exec **represents the runtime of the computation phase.** RandGen **denotes the combined runtime of** BatchRand **and** BatchBundle**.**

7.3.1 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 Fig. 3. We observe that ADprep consistently accounts for the largest proportion of latency, which gradually increases from 41.1% to 54.3% as the committee size increases. The reason ADprep 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 ADprep, we can obtain correct triples while maintaining optimal resilience in asynchronous networks, without relaxing network assumptions during the preprocessing. Compared to ADprep, the latency of the circuit evaluation Exec is significantly smaller. When committee size is 20, the delay is reduced by approximately 86.64s. Therefore, our protocol will perform better in circuits with sparse multiplication gates.

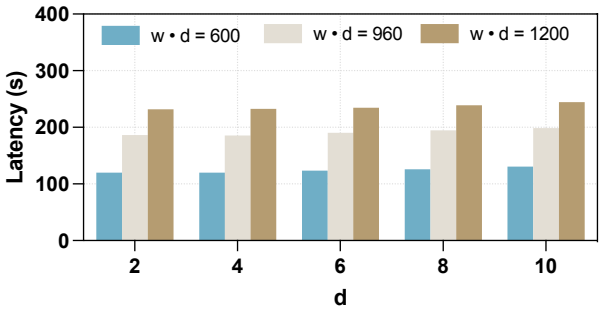


Figure 4: The latency of AD-MPC **under different circuit shapes. We set the number of gates in the circuit to 600, 960 and 1200 (i.e., $w \cdot d = 600, 960, 1200$) and then vary both the depth d and width w simultaneously to test the latency of** AD-MPC **under committee size $n = 16$.**

7.3.2 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 AD-MPC. Fig. 4 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 10.85s, 11.99s, 12.86s, respectively. This trend occurs because, while the total number of gates remains constant, increasing the circuit depth results in more epochs, each incurring a communication overhead of $O(\kappa n^2 \log n \cdot w + \kappa n^3)$. Given that $w \cdot d$ is fixed, the first term stays constant, but the second term $\kappa n^3 \cdot d$ grows linearly with d, driving up the total latency.

7.3.3 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 (AD-MPC, Fluid MPC 1.0, Fluid MPC 2.0 and HoneybadgerMPC). The 7th layer represents the output stage where clients receive and verify shares. Note that Fluid MPC adds an additional layer (the 0th 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 4th layer and observe the protocol's behavior. The results are shown in Fig. 5.

While running Fluid MPC 1.0, we observe that at the 1th layer, different servers in $\mathcal{P}^{1}$ produced six distinct sets (Warning: Inconsistent sets!), leading to the interpolation of six different degree-t

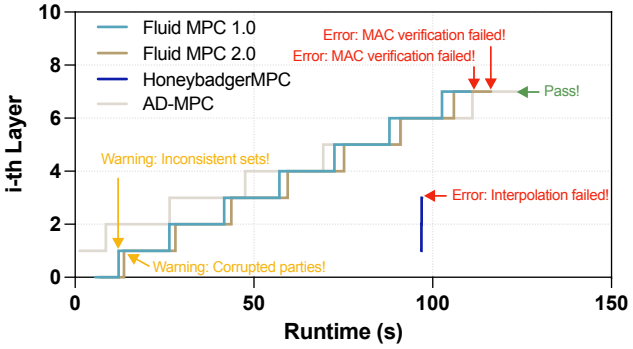


Figure 5: The progress of AD-MPC, Fluid MPC 1.0, Fluid MPC 2.0 and HoneybadgerMPC. We fix the circuit width $w = 100$, the circuit depth $d = 6$, and the committee size $n = 16$. The (7)th layer represents the phase where clients receive shares and attempt to reconstruct them.

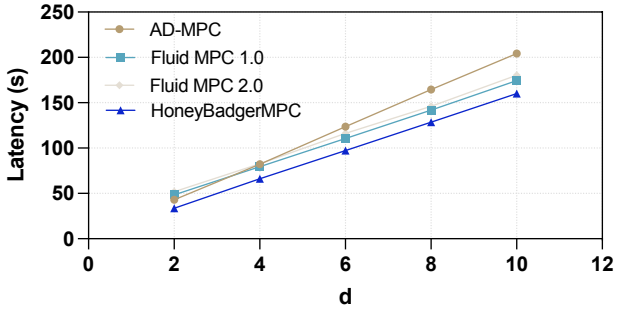


Figure 6: The latency of AD-MPC, Fluid MPC 1.0, Fluid MPC 2.0 and HoneybadgerMPC. We fix the circuit width $w = 100$ and committee size $n = 16$, and change the circuit depth $d = 2, 4, 6, 8, 10$.

polynomials, though their constant terms remain the original secrets. By the 2th 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 2th 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 execute error correction 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 ADprep.

In our main-protocol AD-MPC, although it operates in asynchronous and dynamic settings, it successfully achieves guaranteed output delivery, thus meeting our goals set out in Section 3.1. In comparison, its end-to-end latency exceeds that of Fluid MPC 2.0 by approximately 8.59s.

7.3.4 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 Fig. 6. 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 AD-MPC is steeper.

8 Conclusion

In this paper, we presented AD-MPC, the first asynchronous dynamic MPC protocol that achieves guaranteed output delivery under optimal resilience ($n = 3t + 1$). To address critical challenges inherent in asynchronous dynamic environments, we developed two core techniques: an asynchronous dynamic preprocessing protocol, ADprep, which significantly reduces the overhead for Beaver triple generation to $O(\kappa n^2 \log n)$ per triple; and an asynchronous transfer protocol, ADtrans, which guarantees transfer consistency between successive committees with quasi-quadratic communication complexity. Our implementation demonstrates that AD-MPC achieves performance comparable to the seminal DMPC protocol Fluid MPC, while simultaneously providing stronger security guarantees suitable for asynchronous and dynamic network conditions.

Acknowledgments

We thank Yuan Lu and Yifan Song for their insightful suggestions, and thank the reviewers for their valuable comments that helped improve this paper. This study was supported by the National Key R&D Program of China (No.2022YFB4501000), the National Natural Science Foundation of China (No.62302266, 62232010, U23A20302, U24A20244), Shandong Science Fund for Excellent Young Scholars (No.2023HWYQ-008), and Shandong Science Fund for Key Fundamental Research Project (ZR2022ZD02).

References

[1] Ittai Abraham, Gilad Asharov, Arpita Patra, and Gilad Stern. 2023. Perfectly Secure Asynchronous Agreement on a Core Set in Constant Expected Time. *IACR Cryptol. ePrint Arch.* 2023 (2023), 1130.

[2] Ittai Abraham, Eli Chouatt, Yossi Gilad, Gilad Stern, and Sophia Yakoubov. 2025. Asynchronous Algorand: Reaching Agreement with Near Linear Communication and Constant Expected Time. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*. 28–38.

[3] Anasuya Acharya, Carmit Hazay, Vladimir Kolesnikov, and Manoj Prabhakaran. 2022. Scales: Mpc with small clients and larger ephemeral servers. In *Theory of Cryptography Conference*. Springer, 502–531.

[4] Akhil Bandarupalli, Xiaoyu Ji, Aniket Kate, Chen-Da Liu-Zhang, and Yifan Song. 2024. Concretely Efficient Asynchronous MPC from Lightweight Cryptography. *Cryptology ePrint Archive* (2024).

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

[6] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. 2018. 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*. 695–712.

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

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

[9] Rishabh Bhadauria, James Hsin-yu Chiang, Divya Ravi, Jure Sternad, and Sophia Yakoubov. 2024. Minimizing the Use of the Honest Majority in YOSO MPC with Guaranteed Output Delivery. *Cryptology ePrint Archive* (2024).

[10] Alexander Bienstock, Daniel Escudero, and Antigoni Polychroniadou. 2023. On Linear Communication Complexity for (Maximally) Fluid MPC. *Cryptology ePrint Archive* (2023).

[11] Alexander Bienstock, Daniel Escudero, and Antigoni Polychroniadou. 2024. Perfectly Secure Fluid MPC with Abort and Linear Communication Complexity. *Cryptology ePrint Archive* (2024).

[12] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. 2018. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 315–334.

[13] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. 2002. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. 88–97.

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

[15] Ran Canetti, Sebastian Kolby, Divya Ravi, Eduardo Soria-Vazquez, and Sophia Yakoubov. 2023. Taming Adaptivity in YOSO Protocols: The Modular Way. In *Theory of Cryptography Conference*. Springer, 33–62.

[16] Ignacio Cascudo, Bernardo David, Lydia Garms, and Anders Konring. 2022. YOLO YOSO: fast and simple encryption and secret sharing in the YOSO model. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 651–680.

[17] David Chaum and Torben Pryds Pedersen. 1992. Wallet databases with observers. In *Annual international cryptology conference*. Springer, 89–105.

[18] Arka Rai Choudhuri, Aarushi Goel, Matthew Green, Abhishek Jain, and Gabriel Kaptchuk. 2021. 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*. Springer, 94–123.

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

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

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

[22] Ashish Choudhury and Arpita Patra. 2023. On the Communication Efficiency of Statistically Secure Asynchronous MPC with Optimal Resilience. *Journal of Cryptology* 36, 2 (2023), 13.

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

[24] Ivan Damgård, Daniel Escudero, and Antigoni Polychroniadou. 2021. Phoenix: Secure computation in an unstable network with dropouts and comebacks. *Cryptology ePrint Archive* (2021).

[25] Ivan Bjerre Damgård, Simon Holmgaard Kamp, Julian Loss, and Jesper Buus Nielsen. 2025. Asynchronous YOSO a la Paillier. *Cryptology ePrint Archive* (2025).

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

[27] Sourav Das, Zhuolun Xiang, and Ling Ren. 2021. Asynchronous data dissemination and its applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2705–2721.

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

[29] Bernardo David, Giovanni Deligios, Aarushi Goel, Yuval Ishai, Anders Konring, Eyal Kushilevitz, Chen-Da Liu-Zhang, and Varun Narayanan. 2023. Perfect MPC over layered graphs. In *Annual International Cryptology Conference*. Springer, 360–392.

[30] Giovanni Deligios, Anders Konring, Chen-Da Liu-Zhang, and Varun Narayanan. 2025. Statistical Layered MPC. In *Theory of Cryptography Conference*. Springer, 362–394.

[31] Andreas Erwig, Sebastian Faust, and Siavash Riahi. 2021. Large-scale non-interactive threshold cryptosystems in the yoso model. *Cryptology ePrint Archive* (2021).

[32] Hanwen Feng, Yingzi Gao, Yuan Lu, Qiang Tang, and Jing Xu. 2025. Practical Asynchronous Distributed Key Reconfiguration and Its Applications. *Cryptology ePrint Archive* (2025).

[33] Amos Fiat and Adi Shamir. 1986. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the theory and application of cryptographic techniques*. Springer, 186–194.

[34] Craig Gentry, Shai Halevi, Hugo Krawczyk, Bernardo Magri, Jesper Buus Nielsen, Tal Rabin, and Sophia Yakoubov. 2021. Yoso: You only speak once: Secure mpc with stateless ephemeral roles. In *Annual International Cryptology Conference*. Springer, 64–93.

[35] Vipul Goyal, Chen-Da Liu-Zhang, and Yifan Song. 2024. Towards Achieving Asynchronous MPC with Linear Communication and Optimal Resilience. *Cryptology ePrint Archive* (2024).

[36] Jens Groth and Victor Shoup. 2022. Design and analysis of a distributed ECDSA signing service. *Cryptology ePrint Archive* (2022).

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

[38] Sebastian Kolby, Divya Ravi, and Sophia Yakoubov. 2022. Constant-round YOSO MPC without setup. *Cryptology ePrint Archive* (2022).

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

[40] Chen-Da Liu-Zhang, Elisaweta Masserova, João Ribeiro, Pratik Soni, and Sri AravindaKrishnan Thyagarajan. 2024. Improved YOSO Randomness Generation with Worst-Case Corruptions. *Cryptology ePrint Archive* (2024).

[41] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. 2019. 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*. 887–903.

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

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

[44] Claus-Peter Schnorr. 1989. Efficient identification and signatures for smart cards. In *Conference on the Theory and Application of Cryptology*. Springer, 239–252.

[45] Yuan Su, Yuan Lu, Jiliang Li, Yuyi Wang, Chengyi Dong, and Qiang Tang. 2024. Dumbo-MPC: Efficient Fully Asynchronous MPC with Optimal Resilience. *Cryptology ePrint Archive* (2024).

[46] Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. 2022. hbACSS: How to Robustly Share Many Secrets. In *Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 24–28.

[47] Thomas Yurek, Zhuolun Xiang, Yu Xia, and Andrew Miller. 2023. Long live the honey badger: Robust asynchronous {DPSS} and its applications. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5413–5430.

A Security Analysis of BatchRand

Theorem A.1 (Termination, Consistency, Randomness, and Secrecy of BatchRand). *If all honest servers in $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$ activate* BatchRand*, then the following properties hold with all but negligible probability:*

- *Termination: All honest servers in $\mathcal{P}^{l+1}$ eventually obtain the output* rand_shares $= \{ [r_{pk}]^{l+1} \}_{p \in [t+1], k \in [B]}$.

- *Consistency: All honest servers in $\mathcal{P}^{l+1}$ agree on the same set of* rand_shares. *Moreover, each honest server P_j^{l+1} holds its local shares $\{[r_{pk}]_j^{l+1}\}_{p\in[t+1],k\in[B]}$ consistently with that agreed-upon set.*
- *Randomness: The values $\{r_{pk}\}_{p\in[t+1],k\in[B]}$ are each uniformly random and independent in $\mathbb{Z}_p$.*
- *Secrecy: The PPT adversary that corrupts up to t servers in each committee $\mathcal{P}^l$ or $\mathcal{P}^{l+1}$ cannot learn or bias any honest server's shares beyond what is inherently revealed by the corrupted servers' own shares. In particular, no additional information about $\{r_{pk}\}$ can be inferred from observing honest servers' protocol messages.*

Lemma A.2 (Termination). *All honest servers in $\mathcal{P}^{l+1}$ eventually obtain the output* rand_shares $= \{[r_{pk}]^{l+1}\}_{p\in[t+1],k\in[B]}$.

Proof. The BatchRand protocol involves two main communication components: BACSS and MVBA. By the completeness of BACSS, all honest servers in $\mathcal{P}^{l+1}$ will eventually receive valid shares from at least $2t+1$ dealers in $\mathcal{P}^l$. In addition, by the termination property of MVBA, all honest servers in $\mathcal{P}^{l+1}$ will eventually agree on the same common subset $T \subseteq \mathcal{P}^l$ of size $2t+1$. Once this subset T is fixed, each honest server P_j^{l+1} can locally perform the linear random extraction step to compute its local outputs $\{[r_{pk}]_j^{l+1}\}_{p\in[t+1],k\in[B]}$ without further communication. Consequently, all honest servers in $\mathcal{P}^{l+1}$ obtain rand_shares $= \{[r_{pk}]^{l+1}\}_{p\in[t+1],k\in[B]}$, thereby terminating the BatchRand protocol. □

Lemma A.3 (Consistency). *All honest servers in $\mathcal{P}^{l+1}$ agree on the same set of* rand_shares. *Moreover, each honest server P_j^{l+1} holds its local shares $\{[r_{pk}]_j^{l+1}\}_{p\in[t+1],k\in[B]}$ consistently with that agreed-upon set.*

Proof. By the termination property of MVBA, all honest servers in $\mathcal{P}^{l+1}$ eventually agree on a common subset $T \subseteq \mathcal{P}^l$ of size $2t+1$. For each dealer $P_i^l \in T$, the completeness of BACSS ensures that all distributed shares lie on a valid degree-t polynomial. After collecting these shares from every dealer in T, each honest server P_j^{l+1} locally performs the random extraction step (via the Vandermonde matrix) on the same subset T. This linear transformation guarantees that all honest servers derive the same random values $\{[r_{pk}]^{l+1}\}_{p\in[t+1],k\in[B]}$ satisfying the degree-t structure. Consequently, the local shares $\{[r_{pk}]_j^{l+1}\}_{p\in[t+1],k\in[B]}$ held by each honest server P_j^{l+1} remain consistent with the globally agreed-upon set of random shares. □

Lemma A.4 (Randomness). *The values $\{r_{pk}\}_{p\in[t+1],k\in[B]}$ are each uniformly random and independent in $\mathbb{Z}_p$.*

Proof. In committee $\mathcal{P}^l$, each honest server P_i^l locally samples random values from $\mathbb{Z}_p$. Since these values are chosen independently by each honest dealer and never revealed in the clear, they remain hidden from the adversary. By the secrecy property of Shamir secret sharing (as used in BACSS), any subset of up to t shares observed by a corrupted coalition in $\mathcal{P}^{l+1}$ is indistinguishable from a random distribution over $\mathbb{Z}_p^t$. Then, MVBA guarantees that all honest servers in $\mathcal{P}^{l+1}$ agree on the same common subset $T \subseteq \mathcal{P}^l$ of size $2t+1$. Because at most t dealers in T can be corrupted, the linear random extraction procedure via the Vandermonde matrix combines the random values from $2t+1$ polynomials in a manner that yields $(t+1)$-out-of-$(2t+1)$ uniform randomness. Each extracted share $[r_{pk}]^{l+1}$ is thus guaranteed to be uniformly random in $\mathbb{Z}_p$ and independent of the adversary's view. □

Lemma A.5 (Secrecy). *The PPT adversary that corrupts up to t servers in each committee $\mathcal{P}^l$ or $\mathcal{P}^{l+1}$ cannot learn or bias any honest server's shares beyond what is inherently revealed by the corrupted servers' own shares. In particular, no additional information about $\{r_{pk}\}$ can be inferred from observing honest servers' protocol messages.*

Proof. We establish secrecy via a sequence of games, showing that a real-world adversary cannot distinguish between the actual protocol execution and a simulated one. Let $\mathcal{H}^l$ and $\mathcal{H}^{l+1}$ be the sets of honest servers in committees $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$, respectively. Let $\mathcal{A}^l$ and $\mathcal{A}^{l+1}$ be the corresponding sets of corrupted servers.

Game 0. The adversary corrupts up to t servers in $\mathcal{P}^l$ (forming $\mathcal{A}^l$) and up to t servers in $\mathcal{P}^{l+1}$ (forming $\mathcal{A}^{l+1}$). The honest servers $\mathcal{H}^l \cup \mathcal{H}^{l+1}$ execute the protocol exactly as prescribed by BatchRand. This represents the real-world protocol execution.

Game 1. A simulator impersonates all honest servers in $\mathcal{H}^l \cup \mathcal{H}^{l+1}$. For each honest dealer $P_i^l \in \mathcal{H}^l$, the simulator chooses random polynomials $\phi_i(\cdot)$ in $\mathbb{Z}_p[x]$ as inputs to BACSS_share$_i$. The simulator also runs MVBA precisely as honest parties would. Consequently, any corrupted server $P_j^{l+1} \in \mathcal{A}^{l+1}$ receives the same type of shares $\{([e_i]_j^{l+1}, C_i, \pi_{i,j})\}_{i\in T}$ for an agreed-upon subset $T \subseteq \mathcal{P}^l$, where $[e_i]_j^{l+1} = \phi_i(j)$.

Since the simulator exactly follows the sub-protocol specifications for all honest roles, the adversary's view in **Game 1** is computationally indistinguishable from that in **Game 0**.

Game 2. Here, the simulator modifies how it provides shares to corrupted servers. Instead of running BACSS_share$_i$, the simulator takes as input the polynomial shares $\{[e_i]_j^{l+1}\}_{i\in\mathcal{H}^l, j\in\mathcal{A}^{l+1}}$ that would be observed by corrupted servers. Then, for each $P_i^l \in \mathcal{H}^l$, it selects polynomials $\phi_i'(\cdot)$ consistent with these corrupted shares (so that $\phi_i'(j) = [e_i]_j^{l+1}$) but otherwise sets $\phi_i'(0) = 0$ to fix any remaining degrees of freedom. The simulator sends $(C_i, \phi_i'(j), \pi_{i,j})$ to the corrupted servers, ensuring the distribution matches what corrupted servers expect. For honest servers' shares, the simulator replaces them with ciphertexts encrypting all-zero messages of identical length.

By construction, this simulation neither changes the adversary's overall view (the corrupted servers still receive the same shares $\{[e_i]_j^{l+1}\}$) nor leaks additional information about the actual random polynomials sampled by honest dealers. Indeed, the simulator only needs to know t, n, and the corrupted shares in order to produce a distribution that is indistinguishable from **Game 1** from the adversary's standpoint.

Because BatchRand extracts random values using $(t+1)$-out-of-$(2t+1)$ polynomials (via the Vandermonde matrix), any coalition of at most t corrupted servers does not gain enough information

to reconstruct or bias the resulting random shares. The sequence of games shows that no additional leakage arises from the protocol messages. Hence, BatchRand preserves secrecy against an adversary corrupting up to t servers in each of $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$.

□

B Definition and Analysis of BatchRec and BatchMul

Algorithm 4 Batch reconstruction for $\mathcal{P}^l$

Inputs: $\mathcal{P}^l$ holds $t+1$ secret sharings $([s_0]^l, \cdots, [s_t]^l)$ and and other system parameters
Outputs: $\mathcal{P}^l$ obtains $\texttt{secrets} = (s_0, \cdots, s_t)$

Server P_i^l (in committee $\mathcal{P}^l$):
101: $\texttt{rec_shares}_i \leftarrow \emptyset$, $\texttt{shares}_i \leftarrow \emptyset$, $\texttt{secrets} \leftarrow \emptyset$
102: **for** $j \in [n]$
103: $\quad [\phi(j)]_i^l \leftarrow \sum_{k=0}^{t} [s_k]_i^l \cdot j^k$
104: $\quad$ send $[\phi(j)]_i^l$ to P_j^l
105: **upon** receiving $[\phi(i)]_j^l$ from P_j^l
106: $\quad \texttt{rec_shares}_i \leftarrow \texttt{rec_shares}_i \cup \{[\phi(i)]_j^l\}$
107: $\quad$ **if** $|\texttt{rec_shares}_i| \geq 2t+1$
108: $\quad\quad$ reconstruct $\phi(i)$ via Lagrange interpolation
109: $\quad\quad$ // The robustness of Shamir secret sharing
110: $\quad\quad$ broadcast $\phi(i)$
111: **upon** receiving $\phi(j)$ from P_j^l
112: $\quad \texttt{shares}_i \leftarrow \texttt{shares}_i \cup \{\phi(j)\}$
113: $\quad$ **if** $|\texttt{shares}_i| \geq 2t+1$
114: $\quad\quad$ reconstruct $\phi(x)$ and recover $(s_0, \ldots, s_t)$
115: $\quad\quad$ // Interpolate polynomial $\phi(x) = \sum_{k=0}^{t} s_k \cdot x^k$
116: $\quad\quad \texttt{secrets} \leftarrow (s_0, \ldots, s_t)$
117: **return** secrets

Algorithm 5 Batch multiplication for $\mathcal{P}^l$

Inputs: $\mathcal{P}^l$ holds input sharings $\{[x_k]^l, [y_k]^l\}_{k\in[t+1]}$, triples $\{([a_k]^l, [b_k]^l, [c_k]^l)\}_{k\in[t+1]}$, and other system parameters
Outputs: $\mathcal{P}^l$ obtains $\texttt{mul_outputs} = ([z_1]^l, \cdots, [z_{t+1}]^l)$

Server P_i^l (in committee $\mathcal{P}^l$):
101: $\texttt{mul_outputs} \leftarrow \emptyset$
102: **for** $j \in [t+1]$
103: $\quad [\alpha_j]_i^l \leftarrow [x_j]_i^l - [a_j]_i^l$, $[\beta_j]_i^l \leftarrow [y_j]_i^l - [b_j]_i^l$
104: activate two instances of BatchRec on $\{[\alpha_j]_i^l, [\beta_j]_i^l\}_{j\in[t+1]}$
105: **wait** until they return $\{\alpha_j, \beta_j\}_{j\in[t+1]}$
106: **for** $j \in [t+1]$
107: $\quad [z_j]_i^l \leftarrow \alpha_j \cdot \beta_j + \alpha_j \cdot [b_j]_i^l + \beta_j \cdot [a_j]_i^l + [c_j]_i^l$
108: $\quad \texttt{mul_outputs} \leftarrow \texttt{mul_outputs} \cup \{[z_j]_i^l\}$
109: **return** mul_outputs

Security of BatchRec and BatchMulc. The protocol BatchRec satisfies the following:

- *Termination:* With high probability, all honest servers eventually obtain the output $\texttt{secrets} = (s_0, \ldots, s_t)$.
- *Correctness:* If an honest server P_i^l obtains $\texttt{secrets} = (s_0, \ldots, s_t)$ and another honest server P_j^l obtains $\texttt{secrets} = (s'_0, \ldots, s'_t)$, then $s_k = s'_k$ for all $k \in [0, t]$ with high probability.

The protocol BatchMul satisfies the following:

- *Termination:* With high probability, all honest servers eventually obtain the output $\texttt{mul_outputs} = ([z_1]^l, \ldots, [z_{t+1}]^l)$.
- *Correctness:* If an honest server P_i^l obtains $\texttt{mul_outputs} = ([z_1]_i^l, \ldots, [z_{t+1}]_i^l)$ and another honest server P_j^l obtains $\texttt{mul_outputs} = ([z'_1]_j^l, \ldots, [z'_{t+1}]_j^l)$, then $z_k = z'_k$ for all $k \in [t+1]$ with high probability.

The security of BatchRec and BatchMul has been thoroughly analyzed in [21]. These results show that both protocols remain secure against an adversary corrupting up to t servers in a committee of size $n = 3t+1$. Since these arguments are well-documented in the literature, we omit a detailed security proof here and refer interested readers to [21] for more information.

C Security Analysis of ADprep

Theorem C.1 (Termination, Validity, and Secrecy of ADprep). *If all honest servers in $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$ execute* ADprep*, then, with all but negligible probability, the following properties hold:*

- *Termination: All honest servers in $\mathcal{P}^{l+1}$ eventually obtain the output* $\texttt{triples} = \left\{\left([f_k(\beta)]^{l+1}, [g_k(\beta)]^{l+1}, [h_k(\beta)]^{l+1}\right)\right\}_{k\in[M]}$.
- *Validity: All honest servers in $\mathcal{P}^{l+1}$ agree on the same set of* triples*. Moreover, each honest server P_j^{l+1} holds its local shares $\left([f_k(\beta)]_j^{l+1}, [g_k(\beta)]_j^{l+1}, [h_k(\beta)]_j^{l+1}\right)$ for all $k \in [M]$, where $h_k(\beta) = f_k(\beta) \cdot g_k(\beta)$, consistent with the globally agreed-upon set.*
- *Secrecy: For the output* $\texttt{triples} = \left\{\left([f_k(\beta)]^{l+1}, [g_k(\beta)]^{l+1}, [h_k(\beta)]^{l+1}\right)\right\}_{k\in[M]}$ *the adversary corrupting up to t servers in each of $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$ gains no additional information about $f_k(\beta)$, $g_k(\beta)$, or $h_k(\beta)$ (beyond the fact that $h_k(\beta) = f_k(\beta) \cdot g_k(\beta)$).*

Lemma C.2 (Termination). *All honest servers in $\mathcal{P}^{l+1}$ eventually obtain the output* $\texttt{triples} = \left\{\left([f_k(\beta)]^{l+1}, [g_k(\beta)]^{l+1}, [h_k(\beta)]^{l+1}\right)\right\}_{k\in[M]}$.

Proof. In the ADprep protocol, all n parallel BACSS instances and a single BatchRand instance are activated in parallel. By the completeness property of BACSS, every honest server in $\mathcal{P}^{l+1}$ will eventually receive valid shares from at least $2t+1$ dealers in $\mathcal{P}^l$. Meanwhile, by the termination property of BatchRand, all honest servers in $\mathcal{P}^{l+1}$ also eventually obtain random shares $\left\{[r_i]^{l+1}\right\}_{i\in[n]}$.

To ensure safe usage of these random values, each honest server p_j^{l+1} broadcasts its local share $[r_i]_j^{l+1}$ only after receiving a valid BACSS instance from P_i^l. By the robustness of Shamir secret sharing, each r_i can be robustly reconstructed once enough shares $[r_i]_j^{l+1}$ have been collected, allowing the committee $\mathcal{P}^{l+1}$ to proceed with the standard "sacrifice trick." Concretely, every server in $\mathcal{P}^{l+1}$ activates two consecutive BatchRec calls to verify each

claimed multiplication relation. By the termination of BatchRec, this verification completes for all honest servers.

After verifying correctness, the servers in $\mathcal{P}^{l+1}$ invoke MVBA to agree on a common subset $T \subseteq \mathcal{P}^l$ of size $2t+1$. Finally, each server runs BatchMul to extract the final Beaver triples from these validated dealers. By the termination of BatchMul, all honest servers in $\mathcal{P}^{l+1}$ eventually obtain $\texttt{triples} = \left\{([f_k(\beta)]^{l+1}, [g_k(\beta)]^{l+1}, [h_k(\beta)]^{l+1})\right\}_{k\in[M]}$, completing the protocol. □

Lemma C.3 (Validity). *All honest servers in $\mathcal{P}^{l+1}$ agree on the same set of* triples. *Moreover, each honest server P_j^{l+1} holds its local shares $([f_k(\beta)]_j^{l+1}, [g_k(\beta)]_j^{l+1}, [h_k(\beta)]_j^{l+1})$ for all $k \in [M]$, where $h_k(\beta) = f_k(\beta) \cdot g_k(\beta)$, consistent with the globally agreed-upon set.*

Proof. During the "sacrifice trick", each honest server P_j^{l+1} only broadcasts its random share $[r_i]_j^{l+1}$ after receiving the valid BACSS instance from P_i^l. The completeness of BACSS guarantees that all honest servers in $\mathcal{P}^{l+1}$ eventually receive triple shares from P_i^l. In particular, if P_i^l is corrupted, it cannot wait to learn r_i before activating its BACSS instance, thereby ensuring that r_i remains uniformly random and independent from the adversary's perspective.

This sacrifice procedure follows [22], ensuring that each triple that passes verification satisfies the correct multiplication relation. By the termination property of MVBA, all honest servers in $\mathcal{P}^{l+1}$ eventually agree on the same subset $T \subseteq \mathcal{P}^l$ of size $2t+1$. For every $P_i^l \in T$, the completeness of BACSS ensures that the shared triples lie on valid degree-t polynomials, and the successful sacrifice check confirms that each triple has the correct multiplication relation.

Finally, by the correctness of BatchMul, all honest servers in $\mathcal{P}^{l+1}$ derive the same random Beaver triples:

$$\texttt{triples} = \left\{([f_k(\beta)]^{l+1}, [g_k(\beta)]^{l+1}, [h_k(\beta)]^{l+1})\right\}_{k\in[M]}$$

satisfying both the degree-t structure and the relation $h_k(\beta) = f_k(\beta)\cdot g_k(\beta)$. Consequently, the local shares $\left\{([f_k(\beta)]_j^{l+1}, [g_k(\beta)]_j^{l+1}, [h_k(\beta)]_j^{l+1})\right\}_{k\in[M]}$ held by each honest server P_j^{l+1} remain consistent with this globally agreed-upon set of Beaver triples. □

Lemma C.4 (Secrecy). *For the output* $\texttt{triples} = \left\{([f_k(\beta)]^{l+1}, [g_k(\beta)]^{l+1}, [h_k(\beta)]^{l+1})\right\}_{k\in[M]}$, *the adversary corrupting up to t servers in each of $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$ gains no additional information about $f_k(\beta)$, $g_k(\beta)$, or $h_k(\beta)$ (beyond the fact that $h_k(\beta) = f_k(\beta) \cdot g_k(\beta)$).*

Proof. We establish secrecy via a sequence of games, showing that a real-world adversary cannot distinguish between the actual protocol execution and a simulated one. Let $\mathcal{H}^l$ and $\mathcal{H}^{l+1}$ be the sets of honest servers in committees $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$, respectively. Let $\mathcal{A}^l$ and $\mathcal{A}^{l+1}$ be the corresponding sets of corrupted servers.

Game 0. The adversary corrupts up to t servers in $\mathcal{P}^l$ (forming $\mathcal{A}^l$) and up to t servers in $\mathcal{P}^{l+1}$ (forming $\mathcal{A}^{l+1}$). The honest servers $\mathcal{H}^l \cup \mathcal{H}^{l+1}$ execute the protocol exactly as prescribed by ADprep. This represents the real-world protocol execution.

Game 1. A simulator impersonates all honest servers in $\mathcal{H}^l \cup \mathcal{H}^{l+1}$. For each honest dealer $P_i^l \in \mathcal{H}^l$, the simulator samples random degree-t polynomials $\phi_{ia}(\cdot), \phi_{ib}(\cdot), \phi_{ic}(\cdot)$ and $\phi_{ix}(\cdot), \phi_{iy}(\cdot), \phi_{iz}(\cdot)$ such that $\phi_{ic}(0) = \phi_{ia}(0) \cdot \phi_{ib}(0)$ and $\phi_{iz}(0) = \phi_{ix}(0) \cdot \phi_{iy}(0)$. These polynomials define the triple shares $([a_i], [b_i], [c_i])$ and $([x_i], [y_i], [z_i])$. Meanwhile, the simulator also chooses random polynomials $\phi_i(\cdot)$ for BatchRand. The simulator carries out MVBA (in BatchRand) exactly as the honest parties would. Hence, any corrupted server $P_j^{l+1} \in \mathcal{A}^{l+1}$ receives shares $\{([a_i]_j^{l+1}, [b_i]_j^{l+1}, [c_i]_j^{l+1}), ([x_i]_j^{l+1}, [y_i]_j^{l+1}, [z_i]_j^{l+1})\}_{i\in T}$ and $\{[e_i]_j^{l+1}\}_{i\in T}$, where $T \subseteq \mathcal{P}^l$ is the common subset determined via MVBA, and $[e_i]_j^{l+1} = \phi_i(j)$, $[a_i]_j^{l+1} = \phi_{ia}(j)$, etc.

Since the simulator follows all sub-protocols honestly on behalf of uncorrupted parties, the adversary's view in **Game 1** is computationally indistinguishable from **Game 0**.

Game 2. Because the new interpolated polynomials $f(\cdot), g(\cdot)$ and $h(\cdot)$ satisfy $h(\cdot) = f(\cdot)\cdot h(\cdot)$. Consider $f(\cdot)$ and $g(\cdot)$ are degree-t polynomials requiring $t+1$ shares to interpolate, and considering that the common subset contains at most t corrupted servers from $\mathcal{P}^{l+1}$, $f(\cdot)$, $g(\cdot)$ and $h(\cdot)$ are uniformly ranomd for the adversary. Hence, ADprep preserves secrecy against an adversary corrupting up to t servers in each of $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$.

Next, the simulator modifies how it provides shares to corrupted servers. Rather than genuinely running BACSS_share$_{i1}$ (for the main/sacrificed triples) and BACSS_share$_{i2}$ (for BatchRand) for each honest dealer P_i^l, the simulator randomly selects values $\phi'_{ia}(0)$, $\phi'_{ib}(0), \phi'_{ix}(0), \phi'_{iy}(0), \phi'_i(0)$ and sets $\phi'_{ic}(0) = \phi'_{ia}(0)\cdot\phi'_{ib}(0)$, $\phi'_{iz}(0) = \phi'_{ix}(0)\cdot\phi'_{iy}(0)$. Given the corrupted servers' shares $\{([a_i]_j^{l+1}, [b_i]_j^{l+1}, [c_i]_j^{l+1}), ([x_i]_j^{l+1}, [y_i]_j^{l+1}, [z_i]_j^{l+1})\}_{i\in\mathcal{H}^l,\, j\in\mathcal{A}^{l+1}}$ and $\{[e_i]_j^{l+1}\}_{i\in\mathcal{H}^l,\, j\in\mathcal{A}^{l+1}}$, the simulator interpolates degree-t polynomials $\phi'_{ia}(\cdot), \phi'_{ib}(\cdot), \phi'_{ic}(\cdot)$, $\phi'_{ix}(\cdot), \phi'_{iy}(\cdot), \phi'_{iz}(\cdot)$ and $\phi'_i(\cdot)$ consistent with these corrupted shares. By construction, the corrupted parties see an identical distribution of shares as in **Game 1**, and thus cannot distinguish these simulations from the real protocol.

After finalizing BatchRand and BACSS_share$_{i1}$, each server P_j^{l+1} broadcasts its random share $[r_i]_j^{l+1}$ to reconstruct r_i. This prevents a malicious dealer P_i^l from adaptively choosing its triple polynomials after learning r_i. Then, the committee $\mathcal{P}^{l+1}$ proceeds with BatchRec, MVBA, and BatchMul as specified in ADprep.

Because the newly interpolated polynomials $f(\cdot), g(\cdot), h(\cdot)$ satisfy $h(\cdot) = f(\cdot)\cdot g(\cdot)$, and each is of degree at most t, any coalition of up to t corrupted servers in $\mathcal{P}^{l+1}$ lacks sufficient information to reconstruct or bias the honest shares. Thus, these polynomials appear uniformly random to the adversary. Consequently, ADprep preserves secrecy against an adversary corrupting up to t servers in each of $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$.

□

D Security Proof for hbPolyCommit with Transfer Interface

In this section, we formally prove that our enhanced hbPolyCommit with the transfer evaluation interface securely satisfies the properties of *correctness*, *evaluation binding*, *polynomial binding*, *transfer binding*, and *hiding* under the discrete logarithm assumption in group $\mathbb{G}$. We begin by recalling these properties formally, then proceed to provide rigorous proofs.

Correctness: Given any $SP \leftarrow \mathsf{Setup}(1^\kappa, t)$, any polynomial $\phi(\cdot) \in \mathbb{Z}_p[x]$, and random values $\hat{s}, r, \hat{r} \xleftarrow{\$} \mathbb{Z}_p$, correctness requires that if commitments $\mathbf{C} \leftarrow \mathsf{PolyCom}(SP, \phi(\cdot), \hat{s})$ are honestly generated, then:

- For any evaluation proofs computed via $\mathsf{ProveEval}(SP, i, \mathbf{C}, \phi(i))$, verification via VerifyEval will always succeed.
- For any consistency proofs computed via $\mathsf{ProveConsis}(SP, \phi(\cdot), \hat{s}, r, \hat{r})$, verification via VerifyConsis will always succeed.

Evaluation binding: For any PPT adversary $\mathcal{A}$,

$$\Pr\left[\begin{array}{c} SP \leftarrow \mathsf{Setup}(1^\kappa, t); \\ (\mathbf{C}, i, (\phi(i), \pi_i), (\phi'(i), \pi_i')) \leftarrow \mathcal{A}(SP) : \\ \mathsf{VerifyEval}(SP, i, \mathbf{C}, \phi(i), \pi_i) = 1 \wedge \\ \mathsf{VerifyEval}(SP, i, \mathbf{C}, \phi'(i), \pi_i') = 1 \wedge \\ \phi(i) \neq \phi'(i) \end{array}\right] \leq \varepsilon(\kappa)$$

Polynomial binding: For any PPT adversary $\mathcal{A}$,

$$\Pr\left[\begin{array}{c} SP \leftarrow \mathsf{Setup}(1^\kappa, t); \\ (I_1, I_2, \mathbf{C}, \{\phi(i), \pi_i\}_{i \in I_1 \cup I_2}) \leftarrow \mathcal{A}(SP) \\ \text{where } I_1, I_2 \subset [n], |I_1| = |I_2| = t+1 : \\ \forall i \in I_1 \cup I_2, \\ \mathsf{VerifyEval}(SP, i, \mathbf{C}, \phi(i), \pi_i) = 1 \wedge \\ \phi_1(\cdot) \leftarrow \sum_{i \in I_1} \lambda_i \cdot \phi(i) \wedge \\ \phi_2(\cdot) \leftarrow \sum_{i \in I_2} \lambda_i \cdot \phi(i) \wedge \\ \phi_1(\cdot) \neq \phi_2(\cdot) \end{array}\right] \leq \varepsilon(\kappa)$$

Transfer binding: For any PPT adversary $\mathcal{A}$,

$$\Pr\left[\begin{array}{c} SP \leftarrow \mathsf{Setup}(1^\kappa, t); \\ (\mathbf{C}, \omega, (m, \hat{m}), (m', \hat{m}')) \leftarrow \mathcal{A}(SP) : \\ \mathsf{VerifyConsis}(SP, \mathbf{C}, m, \hat{m}, \omega) = 1 \wedge \\ \mathsf{VerifyConsis}(SP, \mathbf{C}, m', \hat{m}', \omega) = 1 \wedge \\ m \neq m' \end{array}\right] \leq \varepsilon(\kappa)$$

Hiding: Commitments and proofs reveal no additional information about the underlying polynomials beyond public information. Formally, there exists a simulator $(\mathsf{Sim}_1, \mathsf{Sim}_2)$ such that for any PPT adversary $\mathcal{A}$, the following two distributions are indistinguishable:
Real World:

$$\left\{\begin{array}{c} SP \leftarrow \mathsf{Setup}(1^\kappa, t); (\phi(\cdot), I) \leftarrow \mathcal{A}(SP), \\ \text{where } I \subset [n], |I| = t; \\ (\hat{s}, r, \hat{r}) \xleftarrow{\$} \mathbb{Z}_p; \\ \mathbf{C} \leftarrow \mathsf{PolyCom}(SP, \phi(\cdot), \hat{s}); \\ \forall i \in I,\ \pi_i \leftarrow \mathsf{ProveEval}(SP, i, \mathbf{C}, \phi(i)); \\ (m, \hat{m}, \omega) \leftarrow \mathsf{ProveConsis}(SP, \phi(\cdot), \hat{s}, r, \hat{r}) \\ : (\mathbf{C}, m, \hat{m}, \omega, \{\phi(i), \pi_i\}_{i \in I}) \end{array}\right\}$$

Ideal World:

$$\left\{\begin{array}{c} SP \leftarrow \mathsf{Sim}_1(1^\kappa, t); (\phi(\cdot), I) \leftarrow \mathcal{A}(SP) \\ \text{where } I \subset [n], |I| = t; \\ (\mathbf{C}, m, \hat{m}, \omega, \{\pi_i\}_{i \in I}) \leftarrow \mathsf{Sim}_2(SP, \{(i, \phi(i))\}_{i \in I}) \\ : (\mathbf{C}, m, \hat{m}, \omega, \{\phi(i), \pi_i\}_{i \in I}) \end{array}\right\}$$

We now state and prove the main security theorem.

Theorem D.1. *Assuming the hardness of discrete logarithms in $\mathbb{G}$, the enhanced hbPolyCommit with the transfer evaluation interface securely satisfies correctness, evaluation binding, polynomial binding, and hiding with overwhelming probability.*

Proof. We establish each property individually.
Correctness. Follows directly from the correctness of the underlying hbPolyCommit scheme. Honest computation of evaluation and consistency proofs guarantees successful verification.

Evaluation Binding. Evaluation binding requires that once a commitment $\mathbf{C}$ to $\phi(\cdot)$ is fixed, it is infeasible for an adversary to produce two different evaluations at the same point that both pass verification. This property follows from the evaluation binding property of the hbPolyCommit. In order for both evaluations to verify correctly, the adversary would have to break the discrete logarithm assumption, hence the probability is negligible.

Polynomial Binding. Polynomial binding asserts that an adversary cannot open a commitment to two distinct polynomials. In particular, if an adversary produces sets of evaluation proofs on two subsets I_1 and I_2 (each of size $t+1$) that interpolate to two different degree-t polynomials, then with overwhelming probability the two polynomials must be identical. Since a degree-t polynomial is uniquely determined by $t+1$ points, any adversary attempting to open $\mathbf{C}$ to two different polynomials would violate the polynomial binding property of the hbPolyCommit. This holds under the discrete logarithm assumption with only negligible probability.

Transfer Binding. Transfer binding guarantees that no PPT adversary can generate two distinct masked values $m \neq m'$ along with $(\hat{m}, \hat{m}')$ such that both $(m, \hat{m})$ and $(m', \hat{m}')$ are accepted by VerifyConsis under a fixed commitment pair $(\mathbf{C}, \omega)$. We analyze the following two cases:

- $\hat{m} = \hat{m}'$: The verification equation $g_0^m h^{\hat{m}} = \mathbf{C}[0] \cdot \omega$ must also hold for m'. Cancelling common terms, we obtain $g_0^{m-m'} = 1$, which implies $m = m'$ since $\mathbb{G}$ has prime order. This contradicts the adversary's assumption.
- $\hat{m} \neq \hat{m}'$: The two accepted tuples imply $g_0^m h^{\hat{m}} = g_0^{m'} h^{\hat{m}'}$, which breaks the binding property of the Pedersen commitment. This scenario holds only if computing discrete logarithms is tractable in $\mathbb{G}$.

Therefore, under the discrete logarithm assumption, the adversary's success probability is negligible in the security parameter κ.
Hiding. Let the adversary $\mathcal{A}$ choose the polynomial $\phi(\cdot)$ and a set $I \subset [n]$ of indices with $|I| = t$. Intuitively, the adversary knows evaluations of these polynomials at points indexed by I, but does not know the remaining evaluations, particularly the constant term $\phi(0)$, which are kept secret. We construct the simulator's strategy as follows:

- $\mathsf{Sim}_1(1^\kappa, t)$: The simulator runs the setup algorithm $SP \xleftarrow{\$} \mathsf{Setup}(1^\kappa, t)$ and stores the resulting parameters SP for further steps.
- $\mathsf{Sim}_2(SP, \{i, \phi(i)\}_{i \in I})$: Given SP and the evaluations $\{\phi(i)\}_{i \in I}$, the simulator selects uniformly random elements $(\tilde{\phi}(0), \hat{s}, r, \hat{r}) \xleftarrow{\$} \mathbb{Z}_p$. The simulator defines a simulated polynomial $\tilde{\phi}(\cdot)$ that matches the adversary's known evaluations $\{\phi(i)\}_{i \in I}$ and sets the unknown evaluations at points outside I arbitrarily while keeping the constant term as $\tilde{\phi}(0)$. It then honestly computes the simulated commitment $\mathbf{C} \leftarrow \mathsf{PolyCom}(SP, \tilde{\phi}(\cdot), \hat{s})$, generates the corresponding simultated consistency proof

$(m, \hat{m}, \omega) \leftarrow \mathsf{ProveConsis}(SP, \tilde{\phi}(\cdot), \hat{s}, r, \hat{r})$, and produces simulated evaluation proofs $\{\pi_i \leftarrow \mathsf{ProveEval}(SP, i, \mathbf{C}, \tilde{\phi}(i))\}_{i \in I}$.

Formally, the views of adversary $\mathcal{A}$ in the real world and ideal world are defined as follows:
Real World:

$$\left\{ \begin{array}{c} SP \leftarrow \mathsf{Setup}(1^\kappa, t); \\ \mathbf{C} \leftarrow \mathsf{PolyCom}(SP, \phi(\cdot), \hat{s} \xleftarrow{\$} \mathbb{Z}_p); \\ \forall i \in I,\ \pi_i \leftarrow \mathsf{ProveEval}(SP, i, \mathbf{C}, \phi(i)); \\ (m, \hat{m}, \omega) \leftarrow \mathsf{ProveConsis}(SP, \phi(\cdot), \hat{s}, (r, \hat{r}) \xleftarrow{\$} \mathbb{Z}_p) \end{array} \right\}$$

Ideal World:

$$\left\{ \begin{array}{c} SP \xleftarrow{\$} \mathsf{Setup}(1^\kappa, t); \\ \mathbf{C} \leftarrow \mathsf{PolyCom}(SP, \{\phi(i)\}_{i \in I} \cup \tilde{\phi}(0) \xleftarrow{\$} \mathbb{Z}_p, \hat{s} \xleftarrow{\$} \mathbb{Z}_p); \\ \forall i \in I,\ \pi_i \leftarrow \mathsf{ProveEval}(SP, i, \mathbf{C}, \tilde{\phi}(i)); \\ (m, \hat{m}, \omega) \leftarrow \mathsf{ProveConsis}(SP, \{\phi(i)\}_{i \in I} \cup \tilde{\phi}(0), \hat{s}, (r, \hat{r}) \xleftarrow{\$} \mathbb{Z}_p) \end{array} \right\}$$

We now argue indistinguishability. The simulator, given only parameters t, n, and the t shares known to the adversary $\mathcal{A}$, can generate a simulated protocol view that is indistinguishable from $\mathcal{A}$'s real protocol view. This implies that hbPolyCommit with the transfer evaluation interface leaks no additional information about the polynomial $\phi(\cdot)$ beyond what is already implied by the adversary's shares. Therefore, the views in the real and ideal worlds are computationally indistinguishable.

□

E Batch Random Bundle Generation

E.1 Zero-Knowledge Commitment Consistency Proof

We require a non-interactive zero-knowledge proof to demonstrate that a pair of secret values $(s, \hat{s})$ satisfies two conditions: they appear as the constant terms of two committed degree-t polynomials, and they match the secret embedded in a corresponding Pedersen commitment. Formally, given a public statement $(SP, C, \hat{C}, W)$, where $SP = (\mathbb{G}, g, h, g_0, \ldots, g_t)$ and $(C, \hat{C}, W) \in \mathbb{G}^3$. The prover $\mathcal{P}$ holds a private witness $(s, \hat{s}, a_1, \ldots, a_t, b_1, \ldots, b_t, r, \hat{r}) \in \mathbb{Z}_p^{2t+4}$ such that $C = g_0^s \cdot (\prod_{j=1}^t g_j^{a_j}) \cdot h^r$, $\hat{C} = g_0^{\hat{s}} \cdot (\prod_{j=1}^t g_j^{b_j}) \cdot h^{\hat{r}}$, and $W = g_0^s h^{\hat{s}}$. We denote the resulting non-interactive proof system by

$$\sigma \leftarrow \mathsf{ZKCC.Prove}(SP, \phi(\cdot), \hat{\phi}(\cdot), W),$$
$$b \leftarrow \mathsf{ZKCC.Verify}(SP, C, \hat{C}, W, \sigma),$$

where $\phi(\cdot)$ and $\hat{\phi}(\cdot)$ are the degree-t polynomials corresponding to commitments C and $\hat{C}$, respectively, and $b \in \{0, 1\}$ indicates verification success.

This proof can be instantiated via the classical proof of equality of discrete logarithms over multiple bases [17, 44], made non-interactive using the Fiat–Shamir transform [33]. Alternatively, the three linear constraints inherent to this relation can be succinctly encoded within an inner-product argument, thereby enabling efficient Bulletproof-style proofs [12], as utilized by the hbPolyCommit construction [46]. The latter achieves proofs of size $O(\log n)$.

Security properties. The Zero-Knowledge Commitment Consistency (ZKCC) protocol inherits standard security guarantees directly from its underlying zero-knowledge proof constructions [17, 33]. We state these properties formally below.

Theorem E.1. *In the random-oracle model, the ZKCC proof system satisfies the following properties:*

- *Completeness: If $\mathcal{P}$ is honest, then $\mathcal{V}$ will accepts the proof with probability 1.*
- *Proof-of-knowledge: For any PPT adversary $\mathcal{P}^*$, there exists a PPT extractor such that, if the verifier $\mathcal{V}$ accepts the proof returned by $\mathcal{P}^*$ with non-negligible probability, then the extractor can extract a witness that satisfies the statement with non-negligible probability.*
- *Zero-knowledge: For any PPT adversary $\mathcal{V}^*$, there exists a PPT simulator such that, on input a valid statement and a corresponding witness, the simulator's output is computationally indistinguishable from that of $\mathcal{P}$.*

E.2 Adapting BatchRand to BatchBundle

We introduce BatchBundle, a lightweight variant of the BatchRand protocol, for generating random bundles $([r_{pk}], [\hat{r}_{pk}], g_0^{r_{pk}} h^{\hat{r}_{pk}})$. The overall control flow (i.e., n parallel BACSS instances followed by a single MVBA and a randomness extraction) remains unchanged. We only apply the following local modifications:

(1) *Pair sampling and commitment.* For each index k, every dealer samples a random pair $(e_{ik}, \hat{e}_{ik}) \xleftarrow{\$} \mathbb{Z}_p^2$, computes the corresponding Pedersen commitment $w_{ik} = g_0^{eik} h^{\hat{e}_{ik}}$, and jointly shares both values via a single BACSS instance.
(2) *Consistency proof.* Alongside the standard evaluation proofs, the dealer broadcasts a single zero-knowledge proof σ_{ik} that binds $(e_{ik}, \hat{e}_{ik})$ to w_{ik}. Each recipient verifies σ_{ik} once before accepting the dealer's shares.

Security of BatchBundle. We now state the security properties guaranteed by BatchBundle, including *termination*, *consistency*, *randomness*, *secrecy*, and *commitment consistency*.

Theorem E.2 (Termination, Consistency, Randomness, Secrecy and Commitment Consistency of BatchBundle). *If all honest servers in $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$ activate* BatchBundle*, then the following properties hold with all but negligible probability:*

- *Termination: All honest servers in $\mathcal{P}^{l+1}$ eventually obtain* `rand_bundles` $= \left\{ ([r_{pk}]^{l+1}, [\hat{r}_{pk}]^{l+1}, g_0^{r_{pk}} h^{\hat{r}_{pk}}) \right\}_{p \in [t+1], k \in [B]}$.
- *Consistency: All honest servers in $\mathcal{P}^{l+1}$ agree on the same set of* `rand_bundles`*. Moreover, each honest server P_j^{l+1} holds its local shares $\{ [r_{pk}]_j^{l+1}, [\hat{r}_{pk}]_j^{l+1} \}_{p \in [t+1], k \in [B]}$ consistently with that agreed-upon set.*
- *Randomness: The values $\{r_{pk}, \hat{r}_{pk}\}_{p \in [t+1], k \in [B]}$ are each uniformly random and independent in $\mathbb{Z}_p$.*
- *Secrecy: The PPT adversary that corrupts up to t servers in each committee $\mathcal{P}^l$ or $\mathcal{P}^{l+1}$ cannot learn or bias any honest server's shares beyond what is inherently revealed by the corrupted servers' own shares. In particular, no additional information about $\{r_{pk}, \hat{r}_{pk}\}$ can be inferred from observing honest servers' protocol messages.*

Algorithm 6 Batch random bundle generation for $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$

Inputs: $\mathcal{P}^l$ with batch size B and other system parameters
Outputs: committee $\mathcal{P}^{l+1}$ obtains $\texttt{rand_bundles} = \{([r_{pk}]^{l+1}, [\hat{r}_{pk}]^{l+1}, W_{pk})\}_{p\in[t+1],k\in[B]}$.

Server P_i^l (in committee $\mathcal{P}^l$):
// **1. Distribute random bundles via BACSS**
sample $2B$ random secrets $(e_{i1},\cdots,e_{iB},\hat{e}_{i1},\cdots,\hat{e}_{iB}) \xleftarrow{\$} \mathbb{Z}_p$
for $k \in [B]$
  $w_{ik} \leftarrow g_0^{e_{ik}} h^{\hat{e}_{ik}}$
  $\sigma_{ik} \leftarrow \text{ZKCC.Prove}(SP, \phi_{ik}(\cdot), \hat{\phi}_{ik}(\cdot), w_{ik})$
  // s.t. $\phi_{ik}(0) = e_{ik}$, $\hat{\phi}_{ik}(0) = \hat{e}_{ik}$
invoke BACSS_share$_i$ as dealer to share $\{e_{ik}, \hat{e}_{ik}\}_{k\in[B]}$ with broadcasting $\{\sigma_{ik}, w_{ik}\}_{k\in[B]}$

Server P_j^{l+1} (in committee $\mathcal{P}^{l+1}$):
// **2. Agree on $n-t$ valid and completed dealers**
$T_j \leftarrow \emptyset$, $\texttt{bundles}_j \leftarrow \emptyset$, $\texttt{rand_bundles} \leftarrow \emptyset$
activate BACSS_share$_i$ instance for every $i \in [n]$
upon receiving valid $\{([e_{ik}]_j^{l+1}, C_{ik}, \pi_{ik,j}), ([\hat{e}_{ik}]_j^{l+1}, \hat{C}_{ik}, \hat{\pi}_{ik,j}), \sigma_{ik}, w_{ik}\}_{k\in[B]}$ from BACSS_share$_i$
// Validity verified via VerifyEval
if $\text{ZKCC.Verify}(SP, C_{ik}, \hat{C}_{ik}, w_{ik}, \sigma_{ik}) = 1$ for $\forall\, k \in [B]$
  $\texttt{bundles}_j \leftarrow \texttt{bundles}_j \cup \{[e_{ik}]_j^{l+1}, [\hat{e}_{ik}]_j^{l+1}, w_{ik}\}_{k\in[B]}$
  $T_j \leftarrow T_j \cup \{i\}$
  if $|T_j| = 2t+1$
    call MVBA with snapshot T_j as input and T_j as state
    // MVBA's predicate checks if $T_j \supseteq$ input
wait for MVBA output T, w.l.o.g, let $T = \{P_1,\cdots,P_{2t+1}\}$
// **3. Extract random pairs**
for $k \in [B]$
  $[r_{1k},\cdots,r_{(t+1)k}]_j^{l+1} \leftarrow M \cdot \left[[e_{1k},\cdots,e_{(2t+1)k}]_j^{l+1}\right]^\top$
  $[\hat{r}_{1k},\cdots,\hat{r}_{(t+1)k}]_j^{l+1} \leftarrow M \cdot \left[[\hat{e}_{1k},\cdots,\hat{e}_{(2t+1)k}]_j^{l+1}\right]^\top$
  $[W_{1k},\cdots,W_{(t+1)k}] \leftarrow M \cdot \left[w_{1k},\cdots,w_{(2t+1)k}\right]^\top$
  $\texttt{rand_bundles} \leftarrow \texttt{rand_bundles} \cup \{[r_{pk}]_j^{l+1}, [\hat{r}_{pk}]_j^{l+1}, W_{pk}\}_{p\in[t+1]}$
return rand_bundles

- *Commitment Consistency: For every $p \in [t+1]$ and $k \in [B]$, the values output by* BatchBundle *satisfy $W_{pk} = g_0^{r_{pk}} h^{\hat{r}_{pk}}$.*

Termination, consistency, and randomness are inherited directly from the underlying BatchRand. The only additional step introduced in BatchBundle is the broadcast of a Pedersen commitment and its accompanying ZKCC proof, which are verified locally by each recipient. The new security guarantees, namely secrecy and commitment consistency, are established in the subsequent two lemmas.

Lemma E.3 (Secrecy). *The PPT adversary that corrupts up to t servers in each committee $\mathcal{P}^l$ or $\mathcal{P}^{l+1}$ cannot learn or bias any honest server's shares beyond what is inherently revealed by the corrupted servers' own shares. In particular, no additional information about $\{r_{pk}, \hat{r}_{pk}\}$ can be inferred from observing honest servers' protocol messages.*

Proof. We proceed by a sequence of games, following the same structure as in the secrecy proof of BatchRand (see Lemma A.5). **Game 0**, **Game 1**, and **Game 2** are defined exactly as in that proof, and the adversary's view remains computationally indistinguishable across these games.

Game 3. This game reflects the only difference introduced by BatchBundle: the simulator generates simulated ZKCC proofs for all honest dealers, rather than computing them honestly. By Theorem E.1, the output of the simulator is computationally indistinguishable from a real proof. Thus, the adversary's view in **Game 3** is also indistinguishable from that in **Game 2**.

Since secrecy holds after **Game 2** by the argument for BatchRand, and **Game 3** introduces only simulated proofs that preserve indistinguishability, we conclude that BatchBundle ensures secrecy against any PPT adversary corrupting up to t parties in both $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$. □

Lemma E.4 (Commitment Consistency). *Commitment Consistency: For every $p \in [t+1]$ and $k \in [B]$, the values output by* BatchBundle *satisfy $W_{pk} = g_0^{r_{pk}} h^{\hat{r}_{pk}}$.*

Proof. By the completeness of BACSS, all honest servers in $\mathcal{P}^{l+1}$ eventually receive valid shares from at least $2t+1$ dealers in $\mathcal{P}^l$. The termination property of MVBA guarantees that all honest servers agree on a common subset $T \subseteq \mathcal{P}^l$ with $|T| = 2t+1$. For each dealer $P_i^l \in T$ and index k, the proof-of-knowledge property of the ZKCC proof guarantees the existence of witnesses $(e_{ik}, \hat{e}_{ik})$ such that $w_{ik} = g_0^{e_{ik}} h^{\hat{e}_{ik}}$.

Each server in $\mathcal{P}^{l+1}$ computes $([r_{pk}]^{l+1}, [\hat{r}_{pk}]^{l+1})$ by applying a linear transformation to the values $\{([e_{ik}]^{l+1}, [\hat{e}_{ik}]^{l+1})\}_{i\in T}$. Because exponentiation in $\mathbb{G}$ is homomorphic, this transformation preserves the commitment structure, ensuring that the resulting W_{pk} consistently binds $(r_{pk}, \hat{r}_{pk})$. Hence, every bundle returned by BatchBundle satisfies the commitment consistency. □

Complexity of BatchBundle**.** The total communication complexity of BatchBundle remains asymptotically identical to that of BatchRand, totaling $O(\kappa n^2 \log n \cdot B + \kappa n^3)$ bits. Each ZKCC proof incurs an additional cost of size $O(\log n)$, but this overhead is absorbed by the existing $\log n$ factor in the batch complexity and thus does not alter the overall asymptotic bound.

F Security Analysis of ADtrans

Theorem F.1 (Termination, Transfer Consistency, and Secrecy of ADtrans). *If all honest servers in $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$ execute* ADtrans*, then, with all but negligible probability, the following properties hold:*

- *Termination: All honest servers in $\mathcal{P}^{l+1}$ eventually obtain* $\texttt{new_shares} = \{[s_k]^{l+1}\}_{k\in[B]}$.
- *Transfer Consistency: Whenever committee $\mathcal{P}^l$ transfers a secret sharing $[s]^l$, all honest servers in $\mathcal{P}^{l+1}$ reconstruct a new sharing $[s]^{l+1}$ whose constant term equals the same secret s.*
- *Secrecy: For the output* $\texttt{new_shares} = \{[s_k]^{l+1}\}_{k\in[B]}$*, the adversary corrupting up to t servers in each of $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$*

learns no additional information about the secrets $\{s_k\}_{k\in[B]}$ beyond what is inherently revealed by the corrupted servers' own shares.

Lemma F.2 (Termination). *All honest servers in $\mathcal{P}^{l+1}$ eventually obtain* new_shares $= \{[s_k]^{l+1}\}_{k\in[B]}$.

Proof. The ADtrans protocol involves two main communication components: BACSS and MVBA. By the completeness of BACSS, all honest servers in $\mathcal{P}^{l+1}$ will eventually receive valid shares from at least $2t+1$ dealers in $\mathcal{P}^l$. In addition, by the termination property of MVBA, all honest servers in $\mathcal{P}^{l+1}$ will eventually agree on the same common subset $T \subseteq \mathcal{P}^l$ of size $2t+1$. Once this subset T is fixed, each honest server P_j^{l+1} can locally interpolate new share $\{[s_k]_j^{l+1}\}_{k\in[B]}$. Consequently, all honest servers in $\mathcal{P}^{l+1}$ obtain new_shares $= \{[s_k]^{l+1}\}_{k\in[B]}$, thereby terminating the ADtrans protocol. □

Lemma F.3 (Transfer Consistency). *Whenever committee $\mathcal{P}^l$ transfers a secret sharing $[s]^l$, all honest servers in $\mathcal{P}^{l+1}$ reconstruct a new sharing $[s]^{l+1}$ whose constant term equals the same secret s.*

Proof. We instantiate BACSS with hbPolyCommit and its transfer evaluation interface. By MVBA's termination property, all honest servers in $\mathcal{P}^{l+1}$ agree on a common dealer set $T \subseteq \mathcal{P}^l$ of size $2t+1$. From T they robustly reconstruct the masked pairs $(m_k, \hat{m}_k)$ for every $k \in [B]$. Since all dealers in T broadcast identical W_k, servers in $\mathcal{P}^{l+1}$ know W_k is indeed the output of the BatchBundle protocol; by its commitment consistency property, it must satisfy $W_k = g_0^{r_k} h^{\hat{r}_k}$.

Select any subset $T' \subseteq T$ of size $t+1$ whose aggregated commitment $\omega_k = \prod_{i\in T'} (\omega_{ki})^{\lambda_i} = W_k$ for every k. Transfer binding guarantees that the masked shares $\{[m_k]_i^l\}_{i\in T'}$ are tied to $\mathbf{C}_{ki}$; hence the corresponding secret shares in T' are fixed to the correct values. Because the polynomials shared by dealers have degree at most t, the $t+1$ original shares in T' uniquely determine the polynomial. Each server P_j^{l+1} therefore computes its new share via Lagrange interpolation. The resulting sharing $[s_k]^{l+1}$ preserves the constant term s_k, establishing transfer consistency.

□

Lemma F.4 (Secrecy). *For the output* new_shares $= \{[s_k]^{l+1}\}_{k\in[B]}$, *the adversary corrupting up to t servers in each of $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$ learns no additional information about the secrets $\{s_k\}$ beyond what is inherently revealed by the corrupted servers' own shares.*

Proof. We prove secrecy by constructing a sequence of game-based simulations, showing that a real-world adversary cannot distinguish the actual protocol execution of ADtrans from a simulated one. Let $\mathcal{H}^l$ (resp. $\mathcal{H}^{l+1}$) be the honest servers in $\mathcal{P}^l$ (resp. $\mathcal{P}^{l+1}$), and let $\mathcal{A}^l$ (resp. $\mathcal{A}^{l+1}$) be the corrupted servers in each committee.

Game 0. The adversary corrupts up to t servers in $\mathcal{P}^l$ (forming $\mathcal{A}^l$) and up to t servers in $\mathcal{P}^{l+1}$ (forming $\mathcal{A}^{l+1}$). The honest servers $\mathcal{H}^l \cup \mathcal{H}^{l+1}$ execute the protocol exactly as prescribed by ADtrans. This represents the real-world protocol execution.

Game 1. A simulator impersonates all honest servers in $\mathcal{H}^l \cup \mathcal{H}^{l+1}$. For each honest dealer $P_i^l \in \mathcal{H}^l$, the simulator picks a random degree-t polynomial $\phi_i(\cdot)$ in $\mathbb{Z}_p[x]$ with $\phi_i(0) = [s]_i^l$. It uses $\phi_i(\cdot)$ as input to BACSS*_share$_i$ and invokes ProveConsis just as honest parties would. The simulator also runs MVBA faithfully, ensuring that any corrupted server $P_j^{l+1} \in \mathcal{A}^{l+1}$ receives a set of shares $\{[[s]_i]_j^{l+1}\}_{i\in T}$ consistent with an agreed-upon subset $T \subseteq \mathcal{P}^l$, where $[[s]_i]_j^{l+1} = \phi_i(j)$.

Since the simulator follows the protocol specification exactly (from the viewpoint of any uncorrupted party), the adversary's view in **Game 1** is computationally indistinguishable from that in **Game 0**.

Game 2. Here, the simulator modifies how it provides shares to corrupted servers. Rather than genuinely invoking BACSS_share$_i$ for each honest dealer P_i^l, the simulator takes as input the corrupted shares $\{[[s]_i]_j^{l+1}\}_{i\in\mathcal{H}^l, j\in\mathcal{A}^{l+1}}$ that the adversary would see. It then randomly samples a degree-t polynomial $\phi'(\cdot)$. For each $P_i^l \in \mathcal{H}^l$, the simulator interpolates a polynomial $\phi_i'(\cdot)$ consistent with these corrupted shares (so that $\phi_i'(j) = [[s]_i]_j^{l+1}$) and sets $\phi_i'(0) = \phi'(i)$ to fix any remaining degrees of freedom.

From the adversary's perspective, this change does not alter the distribution of the shares it observes. The hiding property of the hbPolyCommit with transfer evaluation interface ensures that no additional information is revealed. Thus, the view of corrupted servers in **Game 2** remains indistinguishable from **Game 1**. Since **Game 1** is also indistinguishable from **Game 0**, it follows that ADtrans preserves secrecy against an adversary corrupting up to t parties in each of $\mathcal{P}^l$ and $\mathcal{P}^{l+1}$.

□

G Security Analysis of AD-MPC

Theorem G.1 (Privacy, and Guaranteed Output Delivery of AD-MPC). AD-MPC *securely realizes the desired properties of ADMPC (as in Definition 3.1).*

Proof. We prove that AD-MPC satisfies properties of ADMPC:

Privacy. During the input stage, secrecy is guaranteed by the underlying protocols: BACSS, BatchRand (Lemmas A.4, A.5), BatchBundle (Lemma E.3), and ADprep (Lemma C.4). In the execution stage, secrets remain hidden during gate evaluation: BatchMul uses pre-shared Beaver triples to securely multiply; ADtrans (Lemma F.4) transfers shares between committees without exposing the underlying secrets; and BatchRand, BatchBundle, and ADprep remain secrecy within each epoch. In the output stage, the protocol again uses ADtrans to securely hand shares to the recipients; only the designated recipients can reconstruct the outputs. Thus, throughout the entire execution, no additional information is leaked to an adversary beyond what could be inferred from the corrupted servers' own shares.

Guaranteed Output Delivery (GOD). During each epoch, the committee first performs a computation phase. Linear gates are evaluated locally (addition/scalar multiplication on shares). Multiplication gates are securely computed with BatchMul, using Beaver triples prepared by the previous epoch via ADprep. By

Lemma C.3, the triples are correct and consistent. Next, a handoff phase moves the newly computed shared values to the next committee. Here, ADtrans (Lemma F.3) enforces that these transfers preserve the same secret values (i.e., the constant terms of the underlying polynomials are unaltered). Additionally, BatchRand, BatchBundle and ADprep ensure fresh randomness and Beaver triples are correctly generated for the next committee (Lemma A.4, Lemma A.5, Lemma E.3, Lemma C.3). Because each epoch completes both phases consistently, the final committee ultimately receives correct output shares, which can be robustly reconstructed. Hence, AD-MPC provides guaranteed output delivery.

Combining these arguments, we conclude that AD-MPC satisfies privacy, and guaranteed output delivery as stipulated by Definition 3.1.

□