RondoMPC: Asynchronous MPC with G.O.D. made More Practical

Xuanji Meng*, Zhaoyang Xie* , Zhaoxin Yang*✉, Sisi Duan*✉, Aggelos Kiayias†✉
*Tsinghua University †The University of Edinburgh
mxj21{xiezhaoyang25}@mails.tsinghua.edu.cn, zhaoxin_yang@mail.tsinghua.edu.cn, duansisi@tsinghua.edu.cn
Aggelos.Kiayias@ed.ac.uk
✉ Corresponding authors

***Abstract*—Asynchronous multi-party computation (AMPC) enables a set of mutually distrustful parties to securely compute any joint function on their private inputs under arbitrary network delays. The guaranteed output delivery (i.e., G.O.D.) property is essential for availability, but can be very challenging to achieve in practice. HoneyBadgerMPC (CCS 2019), the first practical AMPC protocol, follows Beaver's paradigm that generates Beaver triples offline to simplify online computation. One of the underlying building blocks, the random double sharing protocol that generates random double shares, however, implicitly assumes a synchronous network. DumboMPC (Usenix Sec 2025) overcomes the issue by adopting a two-phase workflow where the underlying consensus protocol has to be executed twice, and thus incurs high latency. Velox (CCS 2025) achieves lower latency and weaker cryptographic assumptions at the cost of sacrificing G.O.D. It remains an open question whether there exists a more practical AMPC protocol with the G.O.D. property.**

In this paper, we present RondoMPC, a practical AMPC protocol with G.O.D. with only one phase of consensus. To achieve this goal, we build a practical asynchronous and complete random double sharing (ACRDS) protocol. Our ACRDS protocol supports batching, so a batch of random double shares can be generated simultaneously. Furthermore, ACRDS supports efficient verification of aggregated secret shares, allowing the random double shares to be used for Beaver triple generation using only one phase of consensus. Our experimentation results show that RondoMPC is highly practical, achieving both higher throughput and lower latency for Beaver triple generation compared to existing protocols.

1. Introduction

Multiparty computation (MPC) [1]–[5] allows n parties to compute a function on their private inputs such that an adversary corrupting up to t parties cannot learn any information beyond their private inputs and the output. MPC is an important building block for applications such as anonymous communication [6, 7], decentralized applications [8, 9], privacy-preserving machine learning [10], and replacing trusted setup for cryptographic systems [11].

Depending on the assumptions on the network and the adversary, MPC can be classified into different types. In this work, we are interested in asynchronous MPC (AMPC) with malicious security, where there does not exist an upper bound on message transmission and processing, and the corrupted parties behave arbitrarily. There is a long line of theoretical work on AMPC [12]–[16]. Unlike MPC making weaker assumptions where practical implementations are widely available [17]–[19], practical AMPC protocols and implementations are not well studied.

Review of practical AMPC protocols. HoneyBadgerMPC [6], the first practical AMPC protocol, is based on the framework of generating Beaver triples [24] from Shamir secret sharing [1, 5]. In such a paradigm, there is an offline phase where Beaver triples are generated (to be used for online multiplication), and an online phase where no additional communication is involved. A Beaver triple is in the form of $(\llbracket a \rrbracket_t, \llbracket b \rrbracket_t, \llbracket c \rrbracket_t)$ such that $c = a \cdot b$, where $a, b, c \in \mathbb{F}$, $\mathbb{F}$ is the finite field, and $\llbracket x \rrbracket_t$ denotes a t-out-of-n secret-sharing. To generate Beaver triples, HoneyBadgerMPC uses *random double sharing* [25, 26] that shares $\llbracket r \rrbracket_t, \llbracket r \rrbracket_{2t}$, where r is a random number. The random double sharing protocol used by HoneyBadgerMPC, however, is a synchronous protocol in the semi-honest model (i.e., the adversary still follows the protocol and does not behave arbitrarily). Thus, the offline phase fails to achieve the so-called *guaranteed output delivery* (G.O.D.) property, which is essential in ensuring the availability of the service.

To overcome the issue, DumboMPC [22] presented an offline phase with G.O.D. guarantees. The approach is inspired by a recent work due to Groth and Shoup (GS23) [21]. GS23 uses a *share and re-share* workflow. In the *share* phase, parties invoke n parallel secret sharing instances to share their secrets, and run a consensus protocol to agree on a *common subset*. By the end of this phase, each party obtains $\llbracket a \rrbracket_t$ and $\llbracket b \rrbracket_t$. By locally multiplying the two shares, each party obtains $\llbracket ab \rrbracket_{2t}$. In the second phase, each party invokes a degree-t secret sharing to re-share $\llbracket ab \rrbracket_{2t}$, and all parties run another instance of the consensus protocol to lower the degree from $2t$ to t. Namely, each party obtains the share $\llbracket ab \rrbracket_t$, and a Beaver triple is generated. To further lower the cost of GS23, DumboMPC uses KZG polynomial commitment [27], which assumes trusted setup.

Such a share and re-share workflow involves two phases of secret sharing and consensus. In the asynchronous net-

Protocol	Async.	G.O.D.	Rounds	Message	Communication	Computation	Assumption
BGW [1]	✗	✓	0	$O(n^2)$	$O(\lambda n^2)$	$O(n^2)$	IT
HoneyBadgerMPC [6]	✓	✗	$\mathcal{R}_{\mathsf{RBC}}$	$O(n^2)$	$O(\lambda n)^*$	$\widetilde{O}(n)$	IT
Velox [20]	✓	✗	$4\mathcal{R}_{\mathsf{RBC}} + \mathcal{R}_{\mathsf{ACS}}$	$O(n^2)^\ddagger$	$O(\lambda n)^\ddagger$	$O(n)^\heartsuit$	RO
GS23 [21]$^\triangle$	✓	✓	$2\mathcal{R}_{\mathsf{bACSS}} + 2\mathcal{R}_{\mathsf{ACS}}$	$O(n^3)$	$O(\lambda n^3)$	$O(n^2)$	I^2CDH + RO
DumboMPC [22]$^\triangle$	✓	✓	$2\mathcal{R}_{\mathsf{bACSS}} + 2\mathcal{R}_{\mathsf{MVBA}}$	$O(n^2)$	$O(\lambda n^2)$	$O(n)^\spadesuit$	q-SDH + RO
AD-MPC [23]	✓	✓	$\mathcal{R}_{\mathsf{bACSS}} + 2\mathcal{R}_{\mathsf{MVBA}}$	$O(n^2)^\diamondsuit$	$O(\lambda n^2 \log n)^\diamondsuit$	$O(n^2 \log n)$	DLog + RO
BJK [15]	✓	✓	$2\mathcal{R}_{\mathsf{RA}} + 8\mathcal{R}_{\mathsf{RBC}} + \mathcal{R}_{\mathsf{ACS}}$	$O(n^2)^\dagger$	$O(\lambda n)^\dagger$	$O(n)^\heartsuit$	RO
RondoMPC (this work)	✓	✓	$\mathcal{R}_{\mathsf{bACSS}} + \mathcal{R}_{\mathsf{MVBA}}$	$O(n^2)^\clubsuit$	$O(\lambda n^2)^\clubsuit$	$O(n)^\spadesuit$	CDH + RO

TABLE 1: Comparison with representative prior work. Message and communication complexities are amortized complexities per party. $\mathcal{R}_{\mathsf{X}}$ denotes the round complexity of one invocation of primitive X. Superscript $\triangle$ denotes the bad-case or worst-case cost for protocols that also report a lower good-case/optimistic cost. * The amortized linear communication overhead claimed by HoneyBadgerMPC only works for large batches. † and ‡ follow the amortized per-multiplication claims in the corresponding papers: BJK has $O(\lambda n^4)$ additive overhead, while Velox has $O(\lambda n^3)$ additive overhead. $^\diamondsuit$ denotes the per-triple bound claimed by AD-MPC. $^\clubsuit$ The amortized bound of RondoMPC when one TRIO invocation generates $D = \Theta(n)$ triples. $^\heartsuit$ Lightweight cryptographic operations such as symmetric-key encryption, hash computation, and pseudorandom generation are considered. $^\spadesuit$ Exponentiations and batched pairing checks.

work, such a consensus protocol can be realized via approaches such as asynchronous common subset (ACS) [28, 29], index ACS (IACS) [30], and multivalued validated Byzantine agreement (MVBA) [31]. Unfortunately, such a protocol is not cheap. Also, each secret sharing instance often involves $O(n^2)$ message complexity, and running n parallel instances for AMPC can also be expensive. Thus, the two sequential phases of secret sharing and consensus will unavoidably become the bottleneck, and it is not clear whether they can be further improved. Note that a recent work, Velox [20], puts forward another practical AMPC protocol which has been shown to outperform DumboMPC. However, Velox only achieves fairness, a weaker guarantee than G.O.D.

Thus, an interesting research question we study is:

Can we build an AMPC protocol with G.O.D. with only one phase of secret sharing and consensus?

Our approach. We summarize our results in Table 1. Our starting point is an asynchronous random double sharing protocol that shares $[\![r]\!]_t, [\![r]\!]_{2t}$ efficiently. The sharing of $[\![r]\!]_{2t}$ requires degree-$2t$ secret sharing, which can be very expensive [32]–[34]. Furthermore, when the double sharing is used for Beaver triple generation (or even online multiplication directly), a degree-$2t$ sharing is often required to be reconstructed. Such a reconstruction requires *all* honest parties to participate, which is extremely challenging in an asynchronous network [15, 35].

Following the observation in ATLAS [36], a double sharing can be represented by combining a degree-t sharing with a degree-$2t$ sharing of zero:

$$[\![s]\!]_{2t} = [\![s]\!]_t + [\![0]\!]_{2t}.$$

Our challenge is to realize this known decomposition in the asynchronous malicious setting. We build B1itz, a batched asynchronous complete secret sharing protocol with regular degree-t slots and degree-$2t$ zero slots, and then use the protocol to construct an asynchronous complete random double sharing protocol with only one consensus phase.

The second challenge is that the resulting double sharings must remain useful for later degree reduction. In particular, generating Beaver triples from random double shares requires parties to open masked degree-$2t$ values of the form $ab - r$. We need to ensure that every party can verify each opening share of $ab - r$, as otherwise the computation is incorrect. To do so, we add *coefficient encodings* to B1itz and derive public validation data in ACRDS. Under the computational Diffie-Hellman (CDH) assumption, the verification of the opening share of $ab - r$ can be derived from the validation data of committed a, b, and r polynomials. In this way, we only need one phase of consensus.

We use ACRDS to build our offline Beaver triple generation protocol, TRIO, and use it to build RondoMPC, our AMPC protocol. We evaluate RondoMPC on Amazon AWS using up to 61 virtual machines. Our experimental results show that compared to DumboMPC, the throughput of Beaver triple generation of RondoMPC is up to 46.8% higher, and the latency is up to 3.7× lower.

2. Related Work

Practical AMPC. AMPC was first studied by Ben-Or, Goldwasser, and Wigderson (BGW) [1, 37]. Beaver later introduced the preprocessing paradigm based on multiplication triples [24, 38], which remains the standard approach for practical MPC: expensive correlated randomness is generated offline, and the online phase consumes it to evaluate multiplication gates. As mentioned in the introduction, we follow the framework by Beaver and build a practical offline phase with G.O.D. guarantee.

Random double sharing and ACRDS. Random double sharing [25, 39] is a natural tool for Beaver triple generation. Given $([\![r]\!]_t, [\![r]\!]_{2t})$, parties can mask a degree-$2t$ product ab, open $ab - r$, and then locally obtain a degree-t sharing of ab by adding back $[\![r]\!]_t$. Hyper-invertible-matrix extraction gives an efficient way to produce many such sharings in the synchronous setting.

The main difficulty is making this idea robust in an asynchronous malicious network. The first practical adaptation of HIM-based extraction to asynchrony, DXKR23 [40], provides dual-threshold secret sharing, but later work found that the degree-$2t$ shares are not robustly reconstructible and therefore do not by themselves give G.O.D. for MPC [22]. The main challenge is making degree-$2t$ sharing verifiable, exactly what we address in this work.

ACS, MVBA, and IACS. The concept of asynchronous common subset (ACS) is due to Ben-Or, Canetti, and Goldreich (BCG) in the context of AMPC [41]. As a problem of independent interest, ACS has been extensively studied [12, 42], and recent years have seen many practical constructions. In ACS, parties hold inputs and agree on a common subset of them. A related primitive is multivalued validated Byzantine agreement (MVBA), introduced by Cachin, Kursawe, Petzold, and Shoup [42], where parties agree on one externally valid value. MVBA has been studied extensively in the computational setting [31, 43, 44], and ACS can be reduced to MVBA.

In the signature-free setting, recent protocols show that practical agreement can be built from lightweight cryptography [28, 40, 45]. Das et al. further refined the notion to index ACS (IACS) [30], where parties agree on a set of validated party indices. We can use either MVBA or IACS as our consensus protocol. In our implementation, we use FIN-MVBA [28].

3. Technical Overview

3.1. Review of Existing Practical AMPC Paradigms

AMPC from Beaver triple generation [24] allows one to generate Beaver triples in the offline phase, so the online computation does not involve extra communication. However, generating Beaver triples in an asynchronous network is not easy. Recall that the shares of a Beaver triple are in the form of $([\![a]\!]_t, [\![b]\!]_t, [\![c]\!]_t)$ such that $c = a \cdot b$, where $a, b, c \in \mathbb{F}$ and $\mathbb{F}$ is the finite field. Both $[\![a]\!]_t$ and $[\![b]\!]_t$ can be generated via Shamir secret sharing or its variants. However, if each party locally computes $[\![a]\!]_t \cdot [\![b]\!]_t$, $[\![ab]\!]_{2t}$ will be created. Thus, one needs to lower the degree from $2t$ to t. Such a degree reduction problem is a lot like online multiplication, a problem that Beaver triples aim to solve, creating a circularity issue.

To address the problem, there are two common approaches: Beaver triple generation from re-sharing; Beaver triple generation from random double sharing.

Beaver triple generation from re-sharing. The work due to Groth and Shoup (GS23) [21] introduces the *re-sharing* of $[\![ab]\!]_{2t}$, which draws inspiration from the classic MPC framework by Ben-Or, Wigderson, and Goldwasser (BGW) for online multiplication [1]. In GS23, each party invokes a degree-t asynchronous complete secret sharing (ACSS) to re-share $[\![ab]\!]_{2t}$. To ensure the re-sharing is correct, Pedersen commitment [46] and zk-proof for the product relationship of the Pedersen commitments are used. DumboMPC [22] extends GS23 by providing a more computation-efficient version using KZG polynomial commitment [27], which assumes trusted setup.

Although these works make it possible to build practical Beaver triple generation protocols with G.O.D., the main performance bottleneck comes from the fact that *two sequential phases* of secret sharing and consensus on the common subset are needed, as summarized in the following:

- In phase 1, each party queries an ACSS instance (which can be replaced with batched ACSS, i.e., BACSS) to share a secret, and all parties query an asynchronous common subset (ACS) or multivalued validated Byzantine agreement (MVBA) protocol to agree on a *common subset* of parties that have successfully completed their ACSS instances. Based on the output of ACS/MVBA, $[\![a]\!]_t$ and $[\![b]\!]_t$ are available.
- In phase 2, each party locally computes $[\![ab]\!]_{2t}$, queries an ACSS instance to re-share $[\![ab]\!]_{2t}$, and all parties query ACS/MVBA again to agree on the common subset. Based on this common subset, $[\![ab]\!]_t$ can be computed.

Unfortunately, both secret sharing and the consensus on the common subset are not cheap. For secret sharing that supports batching, $O(n^3)$ message complexity is often required, and either the computational cost or communication cost can be high. Also, even for the most efficient ACS or MVBA protocols [28, 30, 31], the latency of running one instance is at least a few hundred milliseconds on a moderate-size ($<$10 parties) system deployed in a wide-area network.

Beaver triple generation from random double sharing. Random double sharing was introduced by Damgard and Nielsen (DN07) [25] and refined later [26]. The random double sharing is in the form of $[\![r]\!]_t, [\![r]\!]_{2t}$. Based on $[\![ab]\!]_{2t}$ and random double sharing, each party can locally derive:

$$[\![\alpha]\!]_{2t} \leftarrow [\![ab]\!]_{2t} - [\![r]\!]_{2t}$$

and then reconstructs α. After that, each party derives:

$$[\![ab]\!]_t \leftarrow \alpha + [\![r]\!]_t.$$

Building a practical asynchronous approach using the paradigm above is very challenging. This is mainly because reconstructing $[\![\alpha]\!]_{2t}$ requires *all* honest parties to participate. However, in an asynchronous network, each party can only wait for $n - t$ messages before it proceeds to the next phase of the protocol, and one can not know whether these $n - t$ messages are contributed by honest parties unless the opening share of α can be verified. Furthermore, generating random double sharing can be expensive, as degree-$2t$ sharing is more expensive than degree-t sharing [32]–[34].

To our knowledge, existing works are either less practical or sacrifice some guarantees in return for decent performance. For instance, HoneyBadgerMPC [6] assumes that random double sharing is available offline, which can only be achieved when the network is synchronous. Velox [20] builds a practical MPC approach from only lightweight

cryptography (i.e., hash functions and symmetric cryptography), but the protocol can only achieve fairness instead of G.O.D. Furthermore, AMPC protocols with G.O.D. guarantee [15, 35] are shown to achieve linear communication in the circuit size, but the additive overhead can be very high.

3.2. Overview of Our Work

In this work, we propose RondoMPC, a more practical AMPC approach with G.O.D. guarantee with only one phase of consensus, low additive overhead, and no trusted setup assumption. We go for the route of random double sharing to avoid the re-sharing process. The main technical challenges we address are:

- Inspired by ATLAS [36], a synchronous MPC protocol, we generate double shares via a degree-t sharing plus a degree-$2t$ zero sharing. The first challenge is to realize this decomposition in the asynchronous malicious setting with completeness and only one phase of consensus.
- To ensure that α can be reconstructed, each share $[\![\alpha]\!]_{2t}$ needs to be verified, as otherwise the share by a faulty party can easily render the result incorrect. To achieve this goal, both $[\![ab]\!]_{2t}$ and $[\![r]\!]_{2t}$ need to be verified, and the proofs for verification should be merged.

In this work, we build an asynchronous complete random double sharing primitive to generate random double sharings, and augment the output with validation data that makes the opening of α verifiable. We summarize our technical pathway as follows.

Asynchronous complete random double sharing (ACRDS) with one phase of consensus. Our first building block is a random double sharing protocol, which we define as an ACRDS primitive. Recall that random double sharing outputs a degree-t sharing and a degree-$2t$ sharing of the same random value. In a synchronous setting, ATLAS [36] observes that this resource can be represented by combining a degree-t sharing of a secret with a degree-$2t$ sharing of zero. Our goal is to realize this representation in the asynchronous malicious setting, where parties must agree on the same valid dealer set and obtain compatible shares even when some dealers are faulty or delayed.

Namely, if we have a degree-t sharing of some secret s and a degree-$2t$ sharing of a random zero polynomial, then we can obtain a safe degree-$2t$ sharing of s by executing

$$[\![s]\!]_{2t} = [\![s]\!]_t + [\![0]\!]_{2t}.$$

In this way, the degree-t sharing fixes the constant term, while the additional high-threshold zero sharing provides the entropy needed for secure masking.

To instantiate the secret sharing scheme, we build B1itz, an asynchronous complete secret sharing scheme that supports batching. B1itz is built on top of BatchHaven [47], which supports batched degree-$2t$ sharing and proves the batch of polynomials using a batched polynomial commitment mechanism from Bulletproofs [48]. We further modify the context into *regular-slot* sharing of degree t and *high-threshold* sharing of degree $2t$. In this way, each party can share a batch of both $[\![s]\!]_t$ and $[\![0]\!]_{2t}$ efficiently in one secret sharing instance.

In our ACRDS protocol, we run n parallel B1itz instances for parties to share batches of secrets and zeros, run one instance of consensus [28, 30] to agree on a common subset CS, and aggregate the corresponding shares from dealers in CS. Since the degree-$2t$ component is obtained by adding validated zero sharings to the corresponding degree-t secrets, no additional matching proof is needed.

Proving the validity of $[\![\alpha]\!]_{2t}$ on top of B1itz and ACRDS. To ensure that the opening share of α in the form of degree-$2t$ values $\alpha_k = a_k b_k - r_k$, we introduce *coefficient encodings* to B1itz and use them for verification of the shares of α.

Concretely, let $F_{a,k}$, $F_{b,k}$, and $\widetilde{F}_{r,k}$ denote the output polynomials behind $[\![a_k]\!]_t$, $[\![b_k]\!]_t$, and $[\![r_k]\!]_{2t}$. From the coefficient encodings in B1itz, anyone can derive $A_{k,j} = g^{F_{a,k}(j)}$, $B_{k,j} = g^{F_{b,k}(j)}$, and $R_{k,j} = g^{\widetilde{F}_{r,k}(j)}$ for a public party index j.

When P_j claims that $p_k^{(j)}$ is its share of α_k, any party only needs to check the relation $p_k^{(j)} = F_{a,k}(j)F_{b,k}(j) - \widetilde{F}_{r,k}(j)$ in the exponent. For a batch of D openings, they sample random coefficients $\gamma_1, \ldots, \gamma_D$ and verify whether:

$$\prod_{k=1}^{D} e(A_{k,j}^{\gamma_k}, B_{k,j}) = e\left(\prod_{k=1}^{D} (g^{p_k^{(j)}})^{\gamma_k}, g\right) \cdot e\left(\prod_{k=1}^{D} R_{k,j}^{\gamma_k}, g\right),$$

In this way, each opening share of α can be verified, and each party can simply wait until $n - t$ valid opening shares before it reconstructs them. In this way, we only need one phase of consensus to generate Beaver triples and thus achieve the G.O.D. property. Additionally, we can also use the coefficient encodings to simplify the computation of the vanilla BatchHaven and thus make B1itz more computationally efficient.

4. System Model and Building Blocks

4.1. System Model

We consider an asynchronous secure multi-party computation (AMPC) system. The system consists of a fixed set of n parties $\{P_1, \ldots, P_n\}$, connected by authenticated point-to-point channels. The adversary $\mathcal{A}$ may corrupt up to t parties, which we call *faulty* or *corrupted* parties. Non-faulty parties are *honest* or *correct*. We assume an asynchronous network, where there is no upper bound on message processing or transmission delay, but every message sent by an honest sender to an honest receiver is eventually delivered. We assume $t < n/3$, which is optimal for asynchronous MPC.

There are m external clients who provide inputs to the computation. Without loss of generality, we assume $m = n$, i.e., every server P_i holds one input, so that each party both stores a secret and participates in the computation. We assume a standard public-key infrastructure (PKI): each party P_i has a public/private key pair (pk_i, sk_i), and all

public keys are known to all parties. We summarize the main notation in Table 2.

Phases of the system. The computation is divided into an offline phase and an online phase:

- **Offline phase.** The parties jointly execute a triple generation protocol to generate a batch of Beaver triples [24]

$$\{([\![a_j]\!]_t, [\![b_j]\!]_t, [\![c_j]\!]_t)\}_j$$

such that $a_j, b_j, c_j \in \mathbb{F}$ and $c_j = a_j b_j$ for every generated triple.
- **Online phase.** The inputs from the clients are secret-shared. Addition is computed locally where each party locally adds the secret shares. For multiplication, Beaver triples are consumed and no further communication is involved. Since our work focus on improving the offline phase, we omit the details of the online phase in this paper.

Security properties of the triple generation protocol. The offline triple generation protocol must satisfy the following properties.

- **Validity.** For each index j, every honest party that outputs the j-th triple holds a degree-t share of some $a_j, b_j, c_j \in \mathbb{F}$, and the honest parties' output shares are mutually consistent with $c_j = a_j b_j$.
- **Secrecy.** Let $\mathcal{A}$ be any PPT adversary. Let

$$\mathsf{Hist} := \{(a_j, b_j, c_j)\}_{j \in K}$$

denote the set of valid triples that have been reconstructed or otherwise learned by $\mathcal{A}$. A Beaver triple generation protocol satisfies *secrecy* if for every unlearned index $i \notin K$, there exists a negligible function $\mathsf{negl}(\lambda)$ in the security parameter λ such that

$$\left| \Pr\left[\begin{array}{c} (i, a', b') \leftarrow \mathcal{A}(\mathsf{Hist}, \mathsf{View}) : \\ a' = a_i \ \wedge \ b' = b_i \end{array} \right] - \frac{1}{|\mathbb{F}|^2} \right| \leq \mathsf{negl}(\lambda).$$

Here View contains the entire protocol transcript and all public values available to the adversary, including public validation data. Thus the definition is stronger than a scalar-only history definition: even after seeing past opened triples and all public protocol information, $\mathcal{A}$ cannot predict an unlearned triple better than uniform guessing, up to $\mathsf{negl}(\lambda)$.

Security properties of the online computation protocol. The online computation must satisfy the following properties.

- **Privacy.** The adversary learns nothing beyond the final output.
- **Guaranteed Output Delivery (G.O.D.).** Once at least $n - t$ client inputs are accepted, all honest parties eventually output the corresponding computation result.

4.2. Building Blocks

Batched high-threshold asynchronous complete secret sharing (bHACSS). bHACSS is a high-threshold variant of bACSS. A bACSS instance is run by a dealer P_d to share a batch of secrets $\mathbf{s} = (s_1, \ldots, s_L) \in \mathbb{F}^L$ using degree-t Shamir sharing. The protocol consists of a sharing stage and a reconstruction stage. The sharing stage is both verifiable and complete: each party that completes the sharing stage can verify the correctness of the share it received from the dealer, and all honest parties eventually obtain correct and mutually consistent shares of the dealer's secrets. The reconstruction stage consists of multiple reconstruction rounds, where in each round the parties reconstruct one secret previously shared by the dealer.

TABLE 2: Notations used in this work.

Notation	Description
$\mathbb{F}$	Finite field
n	Number of parties
t	Corruption threshold
N	Number of single shares
B	Number of double shares
L	Batch size
D	Number of Beaver triples
$\mathbf{a}$	Vector
$\mathbf{a} \cdot \mathbf{b}$	Inner product of $\mathbf{a}$ and $\mathbf{b}$
$\mathbf{a}[i]$	The i-th element of vector $\mathbf{a}$
M	Vandermonde matrix
$M[i, j]$	The element in the i-th row and j-th column of matrix M
$M[i, :]$	The i-th row of matrix M
$M[:, j]$	The j-th column of matrix M
$\mathcal{A}$	Adversary
P_i	The i-th party
ID.d	Protocol identifier
$[\![x]\!]_t$	Shamir t-sharing of value x
$\mathrm{Rec}(\cdot)$	Reconstruction
$[n]$	Index set $\{1, \ldots, n\}$
$\lvert S \rvert$	The size of the set S
$H(\cdot)$	Hash function
$\mathsf{negl}(\lambda)$	Negligible function in security parameter λ

A secure bACSS protocol satisfies the following properties. We use ID.d to denote the identifier of the instance.

- **Privacy.** If a correct dealer shared $\mathbf{s} = (s_1, \ldots, s_L)$ using ID.d, then before any correct party starts reconstruction for ID.d, $\mathcal{A}$ has no information about $\mathbf{s}$.
- **Liveness.** 1) If the dealer P_d is correct throughout the sharing stage, then all correct parties complete the sharing stage. 2) If at least $t+1$ correct parties validate their partial outputs and start reconstruction for ID.d and round number r, then every correct party P_i reconstructs some z_r^i for ID.d and r.
- **Correctness.** Once $t + 1$ correct parties have completed the sharing stage for ID.d, there exists a fixed value vector $\mathbf{z} = (z_1, \ldots, z_L)$ such that: 1) if the dealer shared $\mathbf{s}$ using ID.d and is correct throughout the sharing stage, then $\mathbf{z} = \mathbf{s}$; and 2) if a correct party P_i reconstructs z_r^i for ID.d and round number r, then $z_r^i = z_r$.
- **Completeness.** If a correct party completes the sharing stage, then there exists a batch of degree-t polynomials $\{R_k(\cdot)\}_{k \in [L]}$ such that $R_k(0) = s'_k$, and each correct party holds the corresponding shares $\{s'_{k,i} = R_k(i)\}_{k \in [L]}$. If the dealer is correct, then $s'_k = s_k$ for all $k \in [L]$.

In bHACSS, the corruption threshold and the reconstruction threshold are decoupled. Let t denote the maximum

number of corrupted parties and let p denote the reconstruction threshold. A secure bHACSS is a dual-threshold variant of bACSS that tolerates any $t < n/3$ and supports reconstruction from any $p < n - t$ parties.

The privacy requirement of bHACSS is accordingly strengthened as follows: if a correct dealer shared $\mathbf{s}$ using ID.d, and at most $p - t$ correct parties have started reconstruction for ID.d, then $\mathcal{A}$ has no information about $\mathbf{s}$. In terms of concrete constructions, our protocol builds on BatchHaven [47], which relies on batched polynomial commitments derived from Bulletproofs [48].

Multivalued validated Byzantine Agreement (MVBA) over an index. MVBA is defined by *propose* and *decide* events. The value each replica *proposes* and *decides* satisfies a global predicate Q known to all parties. The validation proof can be constructed using digital signatures. Some recent constructions also provide the so-called signature-free MVBA, which $Q()$ depends on some local state of replicas [28]. An MVBA protocol with predicate Q should satisfy the following properties:

- **External validity**: Every correct replica that terminates *decides* v such that $Q(v)$ holds.
- **Agreement**: If a correct replica *decides* v, then any correct replica that terminates will *decides* v.
- **Integrity**: If all replicas follow the protocol, and if a correct replica *decides* v such that $Q(v)$ holds, then some replica *proposed* v such that $Q(v)$ holds.
- **Termination**: If all correct replicas are activated and all messages sent among correct replicas have been delivered, then all correct replicas will *decide*.

In this work, we use FIN-MVBA [28] to agree on an index of n bits. We can also use index asynchronous common subset (IACS) [30] where the contexts are slightly different.

Asynchronous complete random Double sharing (ACRDS). We adopt the definition of random double sharing abstraction from prior work [22, 25, 39], and expose a generalized interface parameterized by two integers (N, B). An instance $\mathsf{ACRDS}(N, B)$ outputs a batch of $N + B$ random secrets $\{r_k\}_{k=1}^{N+B}$ in shared form. Namely, each party P_i outputs

$$\left\{[\![r_k]\!]_t^{(i)}\right\}_{k=1}^{N+B} \quad \text{and} \quad \left\{[\![r_k]\!]_{2t}^{(i)}\right\}_{k=N+1}^{N+B}.$$

Our implementation also outputs public validation data vdata for these shared values. We call the first N entries *single shares*, which are only t-shared, and the last B entries *double shares*, which are both t- and $2t$-shared. Setting $B = 0$ yields a pure batch of N single shares; setting $N = 0$ yields a batch of B double shares.

Note that classical formulations produce either (i) $\Theta(n)$ single shares via linear extraction [22, 25], or (ii) $n - 2t$ double shares with perfect security via a hyper-invertible matrix and public checks [39]. Our (N, B) notion covers both: choosing $(N, B) = (N, 0)$ recovers the single shares, and choosing $(N, B) = (0, B)$ recovers the double shares.

Based on the refined notion, we define the security of ACRDS as follows.

- **Termination.** Every honest party eventually outputs its shares $\{[\![r_k]\!]_t^{(i)}\}_{k=1}^{N+B}$ and $\{[\![r_k]\!]_{2t}^{(i)}\}_{k=N+1}^{N+B}$.
- **Consistency.** For each $k \in [N + B]$, all honest parties' t-shares are consistent: for any two sets of honest parties $\mathcal{H}, \mathcal{H}'$ with $|\mathcal{H}|, |\mathcal{H}'| \geq t + 1$,

$$\mathrm{Rec}\left(\{[\![r_k]\!]_t^{(i)}\}_{i\in\mathcal{H}}\right) = \mathrm{Rec}\left(\{[\![r_k]\!]_t^{(i)}\}_{i\in\mathcal{H}'}\right) = r_k,$$

  where Rec denotes a reconstruction function. Moreover, for each $k \in \{N + 1, \ldots, N + B\}$, the $2t$-shares agree with the t-shares on the same value.
- **Completeness.** If a correct party outputs in an $\mathsf{ACRDS}(N, B)$ instance, then there exists a fixed batch of output secrets $(r_1, \ldots, r_{N+B})$ such that every correct party eventually outputs shares of the same batch:

$$\left\{[\![r_k]\!]_t^{(i)}\right\}_{k=1}^{N+B} \quad \text{and} \quad \left\{[\![r_k]\!]_{2t}^{(i)}\right\}_{k=N+1}^{N+B}.$$

- **Secrecy.** For any PPT adversary $\mathcal{A}$ corrupting a set of parties $C \subseteq [n]$ with $|C| \leq t$, let view_C denote the entire view of corrupted parties in an execution of $\mathsf{ACRDS}(N, B)$, including their internal states, all messages delivered to or from them, and all public values. The protocol satisfies secrecy if for every $j \in [N + B]$,

$$\left|\Pr\left[(j, x) \leftarrow \mathcal{A}(\mathsf{view}_C) : x = r_j\right] - \frac{1}{|\mathbb{F}|}\right| \leq \mathsf{negl}(\lambda),$$

  where r_j is the j-th ACRDS output.

Vandermonde matrix. A Vandermonde matrix is a matrix defined by a sequence of field elements $\alpha_1, \ldots, \alpha_m \in \mathbb{F}$. Given positive integers d and m, its $(d \times m)$ form is

$$V = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_m \\ \alpha_1^2 & \alpha_2^2 & \cdots & \alpha_m^2 \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{d-1} & \alpha_2^{d-1} & \cdots & \alpha_m^{d-1} \end{pmatrix}.$$

Equivalently, $V[\ell, j] = \alpha_j^{\ell-1}$ for $\ell \in [d]$ and $j \in [m]$.

A fundamental property of Vandermonde matrices is that when $\alpha_1, \ldots, \alpha_m$ are pairwise distinct, every square submatrix formed by distinct columns is nonsingular. In particular, if $d = m$, then

$$\det(V) = \prod_{1\leq i<j\leq m} (\alpha_j - \alpha_i) \neq 0.$$

Thus, a square Vandermonde matrix over distinct field points is invertible. Vandermonde matrices are widely used to describe polynomial evaluations and linear transformations induced by distinct evaluation points. In secret sharing related protocols, they are often used to combine multiple sharings while preserving the underlying linear structure.

5. B1itz: BatchHaven with Mixed-Threshold Slots

B1itz is a bHACSS protocol that supports two types of sharings in the same batched sharing instance. Let *slots* be the number of secrets one B1itz protocol shares. For batch parameters N and B, we distinguish two types of slots as follows.

- *regular slots* are degree-t sharings of N secrets, denoted as $(s_1, \ldots, s_N)$;
- *high-threshold slots* are degree-$2t$ sharings. In B1itz, degree-$2t$ sharings only share zeros, denoted as $(0, \ldots, 0)$.

We build B1itz by extending BatchHaven [47], a bHACSS protocol. In BatchHaven, the dealer commits to a batch of sharing polynomials and proves the consistency of all the polynomial evaluations using batched polynomial commitment from Bulletproofs [48]. The proof generation and verification functions are called $\mathrm{BatchEval}$ and $\mathrm{BatchVerifyEval}$, respectively. Compared to earlier ACSS protocols supporting batching, the batched polynomial commitment approach taken by BatchHaven has been shown to be highly practical in the computational setting. We repeat the details of $\mathrm{BatchEval}$ and $\mathrm{BatchVerifyEval}$ of BatchHaven in Appendix A.

For B1itz, we extend BatchHaven to additionally support the degree-$2t$ sharing of zeros. Supporting the sharing of zeros can be easily done via the batched polynomial commitment schemes. We add on top *coefficient encodings* for every sharing polynomial. The coefficient encodings are used mainly when B1itz is used to build our ACRDS protocol. Meanwhile, it allows parties to cheaply verify the public slot layout by inspecting encoded coefficients: regular slots must have zero coefficients in degrees $t+1, \ldots, 2t$, and high-threshold slots must have zero constant coefficients. Thus, verifying the validity of the degree-t sharing and double sharing can be cheap.

The coefficient encodings also simplify receiver-specific consistency checks. In BatchHaven, this consistency is checked through an additional family of T polynomials, together with a batched commitment and evaluation proof for them. In B1itz, once the encodings are bound to the batched commitment, parties can derive the expected value $g^{F_k(x)}$ locally and compare it with the encoded scalar share. This removes the extra T-polynomial route and saves both computation and communication in the sharing, echo, and reconstruction checks.

Additionally, different from BatchHaven where the secrets have to be reconstructed one by one, B1itz can reconstruct multiple slots of secrets at a time. Specifically, since the reconstruction points are shared across all target slots, parties compute the Lagrange basis once and reuse it for all recovered values. This batch reconstruction reduces the interpolation work compared with reconstructing slots independently, and directly speeds up the later Beaver-triple generation step where many masked degree-$2t$ products are opened together.

The workflow. As shown in Figure 1, the sharing stage of the B1itz protocol consists of three phases: share, echo, and ready. As mentioned above, we build coefficient encodings that will later be needed only when B1itz is used to build our ACRDS protocol. We thus highlight the coefficient-encoding and pairing-consistency procedures in blue. Thus, the workflow without any highlighted colors is the minimum workflow to build a secure bHACSS protocol. Below, we first describe the main workflow and then discuss the procedure for our coefficient encodings.

- **The share phase.** In Lines 03–05, the dealer samples N regular degree-t sharing polynomials $F_k^{\mathsf{reg}}(X)$ and B high-threshold degree-$2t$ zero-sharing polynomials $F_k^{\mathsf{hi}}(X)$, and commits to the whole batch using $\mathrm{BatchCommit}$ from Figure 2 by generating $\widehat{\mathcal{F}}$. The $\mathrm{BatchCommit}$ function generates a vector of Bulletproof-style commitments for both $F_k^{\mathsf{reg}}(X)$ and $F_k^{\mathsf{hi}}(X)$. In Lines 07–08, the dealer prepares a degree-t polynomial $S_{k,j}$ for each $k \in [N+B]$ and j, where $S_{k,j}(j) = F_k(j)$, and commits to the whole S batch. In Lines 09–13, the dealer evaluates the S batch and sends each party its receiver-specific evaluation vector and proof. The relation between the delivered scalar shares and the committed recovery polynomials is verified later by LinkCheck using the coefficient encodings.
- **The echo phase.** In Lines 14–18, upon receiving a $(\mathrm{ID}.d, \mathsf{send}, \mathsf{set}_i)$ message, each party verifies that the root commitment C is valid, checks uncached global validation data, and uses $\mathrm{BatchVerifyEval}$ to validate its received evaluations. It then invokes LinkCheck to check that the scalar share vector certified by $\mathrm{BatchVerifyEval}$ is equal to the evaluation of the committed batch $\mathcal{F}$ at point i. Note that party P_i receives evaluations of all $S_{k,j}$ at point i. If all checks pass, in Lines 19–20, each party prepares an $\mathsf{info}_{i,j}$ data for each $j \in [n]$, consisting of the evaluations of the polynomials $\{S_{k,j}\}_k$ at point i, and the corresponding commitments and proofs, and sends $\mathsf{info}_{i,j}$ to P_j.
- **The ready phase.** In Lines 21–25, a party verifies receiver-specific echo messages and stores the valid ones. The global coefficient and slot checks are reused from the cache whenever possible; the receiver-specific $\mathrm{BatchVerifyEval}$ check is still performed for every new echo point. In Lines 26–29, the standard *ready amplification* [49] is used: a party sends a (ready) message after collecting enough valid echo messages or $t+1$ matching (ready) messages. In Lines 30–32, once $2t+1$ (ready) messages and enough valid echo data are available, the party obtains its scalar shares, reruns LinkCheck on the final share vector, and outputs valid regular shares, high-threshold zero shares, and validation data. Here, validation data denotes the checkable transcript attached to the scalar shares, i.e., the commitment, encoding, evaluation, and proof objects carried in set_i.

Coefficient encoding. As mentioned above, we build a coefficient encoding mechanism in the protocol that is not a must for the correctness of the protocol. Instead, it is crucial when B1itz is used to build our ACRDS protocol. We now

▷ **B1itz Sharing Stage (Code for P_i)**
Input: regular values $\mathbf{s} = (s_1, \ldots, s_N)$ and batch parameters (N, B).
Output: local regular shares $\{[\![s_k]\!]_t^{(i)}\}_{k\in[N]}$, local high-threshold shares $\{[\![0]\!]_{2t}^{(i)}\}_{k\in[B]}$, and validation data.
Notation: $L = N + B$.

▷ **Initialization.**
01: $valid_i \leftarrow \mathsf{false}$, $ready_i \leftarrow \mathsf{false}$, $E_i \leftarrow \emptyset$, and $G_i \leftarrow \emptyset$.

▷ **The share phase.**
02: Upon receiving $(\mathrm{ID}.d, \mathsf{in}, \mathsf{share}, \mathbf{s}; N, B)$, as a dealer (i.e., $P_i = P_d$):
03: For $k \in [N]$, sample a degree-t polynomial $F_k^{\mathsf{reg}}(X)$ such that $F_k^{\mathsf{reg}}(0) = s_k$; pad its coefficient vector to length $2t+1$.
04: For $k \in [B]$, sample a degree-$2t$ polynomial $F_k^{\mathsf{hi}}(X)$ such that $F_k^{\mathsf{hi}}(0) = 0$.
05: Let $\mathcal{F} \leftarrow \{F_k^{\mathsf{reg}}\}_{k\in[N]} \cup \{F_k^{\mathsf{hi}}\}_{k\in[B]}$. Compute $\widehat{\mathcal{F}} \leftarrow \mathrm{BatchCommit}(crs, \mathcal{F}, 2t)$.
06: For each $F(X) = \sum_{\ell=0}^{2t} f_\ell X^\ell \in \mathcal{F}$, compute its coefficient encoding

$$\mathsf{Enc}(F) \leftarrow (g^{f_0}, g^{f_1}, \ldots, g^{f_{2t}}).$$

Let $\mathsf{Enc}(\mathcal{F})$ denote the list of all such encodings.
07: For $k \in [L], j \in [n]$, sample a degree-t polynomial $S_{k,j}(X)$ such that $S_{k,j}(j) = F_k(j)$.
08: Compute $\hat{S} \leftarrow \mathrm{BatchCommit}(crs, \{S_{k,j}\}_{k\in[L],j\in[n]}, t)$.
09: Set $C \leftarrow \mathrm{vCom}(\widehat{\mathcal{F}}, \hat{S})$.
10: Compute $(Y^S, \Phi^S) \leftarrow \mathrm{BatchEval}(crs, \{S_{k,j}\}_{k\in[L],j\in[n]}, \hat{S}, (1, \ldots, n), t)$.
11: For each $j \in [n]$, set $\vec{y}_j^S \leftarrow Y^S[:, j]$ and $\phi_j^S \leftarrow \Phi^S[j]$.
12: For each $j \in [n]$, set $\mathsf{set}_j \leftarrow \{C, \widehat{\mathcal{F}}, \hat{S}, \mathsf{Enc}(\mathcal{F}), \vec{y}_j^S, \phi_j^S\}$.
13: For each $j \in [n]$, send $(\mathrm{ID}.d, \mathsf{send}, \mathsf{set}_j)$ to P_j.

▷ **The echo phase.**
14: Upon receiving $(\mathrm{ID}.d, \mathsf{send}, \mathsf{set}_i)$ from P_d for the first time, where $\mathsf{set}_i = \{C, \widehat{\mathcal{F}}, \hat{S}, \mathsf{Enc}(\mathcal{F}), \vec{y}_i^S, \phi_i^S\}$:
15: Verify that $C = \mathrm{vCom}(\widehat{\mathcal{F}}, \hat{S})$.
16: If $(C, \widehat{\mathcal{F}}, \mathsf{Enc}(\mathcal{F})) \notin G_i$, verify that $\mathsf{CoeffCheck}(\widehat{\mathcal{F}}, \mathsf{Enc}(\mathcal{F})) = \mathsf{true}$ and $\mathsf{SlotCheck}(\mathsf{Enc}(\mathcal{F})) = \mathsf{true}$; then add it to G_i.
17: Verify that $\mathrm{BatchVerifyEval}(crs, \hat{S}, \vec{y}_i^S, (i), t; \phi_i^S) = \mathsf{true}$.
18: Verify that $\mathsf{LinkCheck}(\mathsf{Enc}(\mathcal{F}), \{\vec{y}_i^S[k, i]\}_{k\in[L]}, i) = \mathsf{true}$.
19: For $k \in [L], j \in [n]$, compute $\hat{S}_{new}$, y_{new}, and ϕ_{new}: $\hat{S}_{new}[k] \leftarrow \hat{S}[k, j]$, $y_{new}[k, j] \leftarrow \vec{y}_i^S[k, j]$, and $\phi_{new}[k, j] \leftarrow \phi_i^S[k, j]$.
20: If all checks pass, set $valid_i \leftarrow \mathsf{true}$. For each $j \in [n]$, send $(\mathrm{ID}.d, \mathsf{echo}, \mathsf{info}_{i,j})$ to P_j, where $\mathsf{info}_{i,j} = \{C, \widehat{\mathcal{F}}, \hat{S}_{new}, \mathsf{Enc}(\mathcal{F}), y_{new}[:, j], \phi_{new}[:, j]\}$.

▷ **The ready phase.**
21: Upon receiving $(\mathrm{ID}.d, \mathsf{echo}, \mathsf{info}_{j,i})$ from P_j for the first time:
22: Parse $C, \widehat{\mathcal{F}}, \hat{S}_{new}, \mathsf{Enc}(\mathcal{F})$, $y_{new}[:, i]$, and $\phi_{new}[:, i]$ from $\mathsf{info}_{j,i}$, and verify that $\widehat{\mathcal{F}}$ and all entries of $\hat{S}_{new}$ are in C.
23: If $(C, \widehat{\mathcal{F}}, \mathsf{Enc}(\mathcal{F})) \notin G_i$, verify that $\mathsf{CoeffCheck}(\widehat{\mathcal{F}}, \mathsf{Enc}(\mathcal{F})) = \mathsf{true}$ and $\mathsf{SlotCheck}(\mathsf{Enc}(\mathcal{F})) = \mathsf{true}$; then add it to G_i.
24: Verify that $\mathrm{BatchVerifyEval}(crs, \hat{S}_{new}, y_{new}[:, i], (j), t; \phi_{new}[:, i]) = \mathsf{true}$.
25: If all checks pass, add the echo message to E_i.
26: If $|E_i| \geq 2t + 1$ for the same C and $ready_i = \mathsf{false}$, set $ready_i \leftarrow \mathsf{true}$ and send $(\mathrm{ID}.d, \mathsf{ready}, C)$ to all parties.
27: Upon receiving $(\mathrm{ID}.d, \mathsf{ready}, C)$ from P_j for the first time:
28: If $t + 1$ ready messages with the same C are received and $ready_i = \mathsf{false}$, set $ready_i \leftarrow \mathsf{true}$ and send $(\mathrm{ID}.d, \mathsf{ready}, C)$ to all parties.
29: If $2t + 1$ ready messages with the same C are received, wait until $|E_i| \geq t + 1$.
30: If $valid_i = \mathsf{true}$, set $V_i \leftarrow \{\vec{y}_i^S[k, i]\}_{k\in[L]}$. Otherwise, interpolate $\{S_{k,i}\}_{k\in[L]}$ from the valid echo messages in E_i and set $V_i \leftarrow \{S_{k,i}(i)\}_{k\in[L]}$.
31: Verify that $\mathsf{LinkCheck}(\mathsf{Enc}(\mathcal{F}), V_i, i) = \mathsf{true}$.
32: Output $(\mathrm{ID}.d, \mathsf{out}, \mathsf{shared}, \mathsf{set}_i)$ with the scalar shares V_i and validation data.

Figure 1: The sharing stage of B1itz for node P_i and tag $\mathrm{ID}.d$. The workflow for coefficient encoding is highlighted in blue.

func $\mathrm{BatchCommit}(crs, \mathcal{F}, d)$:
01: For each $F_k(X) = \sum_{\ell=0}^{d} f_{k,\ell} X^\ell \in \mathcal{F}$, compute

$$\widehat{\mathcal{F}}[k] \leftarrow \prod_{\ell=0}^{d} g_\ell^{f_{k,\ell}}.$$

02: Output $\widehat{\mathcal{F}}$.

Figure 2: The $\mathrm{BatchCommit}$ function of B1itz.

discuss the procedure.

In the share phase (Line 06 of Figure 1), the dealer publishes coefficient encodings $\mathsf{Enc}(\mathcal{F})$ for each polynomial in the batch. In the echo phase and the ready phase, each party checks coefficient consistency via the CoeffCheck function, slot validity via the SlotCheck function, and receiver-specific linkage via the LinkCheck function. The first two checks depend only on the global validation data and can be cached after one successful verification. We now discuss these functions in detail.

- CoeffCheck. This check proves that $\mathsf{Enc}(\mathcal{F})$ and $\widehat{\mathcal{F}}$ encode the same coefficient vectors. Since $\widehat{\mathcal{F}}$ is a vector of Bulletproof-style commitments using independent bases

▷ **B1itz Reconstruction Stage (Code for P_i)**
Input: target slot set $\mathcal{I}$, local sharing output $(\mathrm{ID}.d, \mathsf{out}, \mathsf{shared}, \mathsf{set}_i)$, and validation data.
Output: reconstructed values for slots in $\mathcal{I}$.

01: $rec_i \leftarrow \mathsf{false}$, $R_i \leftarrow \emptyset$, and reuse the validation cache G_i from the sharing stage.
02: Upon receiving $(\mathrm{ID}.d, \mathsf{in}, \mathsf{reconstruct}, \mathcal{I})$:
03: Parse set_i as $C, \widehat{\mathcal{F}}, \hat{S}, \mathsf{Enc}(\mathcal{F}), \vec{y}_i^S, \phi_i^S$.
04: Set $d_{\mathcal{I}} \leftarrow t$ if every $idx \in \mathcal{I}$ is a regular-slot index, and $d_{\mathcal{I}} \leftarrow 2t$ otherwise.
05: Send $(\mathrm{ID}.d, \mathsf{reconstruct\text{-}share}, \mathcal{I}, C, \widehat{\mathcal{F}}, \hat{S}, \mathsf{Enc}(\mathcal{F}), \vec{y}_i^S, \phi_i^S)$ to all parties.
06: Upon receiving $(\mathrm{ID}.d, \mathsf{reconstruct\text{-}share}, \mathcal{I}, C, \widehat{\mathcal{F}}, \hat{S}, \mathsf{Enc}(\mathcal{F}), \vec{y}_j^S, \phi_j^S)$ from P_j for the first time:
07: Verify that $C = \mathrm{vCom}(\widehat{\mathcal{F}}, \hat{S})$.
08: If $(C, \widehat{\mathcal{F}}, \mathsf{Enc}(\mathcal{F})) \notin G_i$, verify that $\mathsf{CoeffCheck}(\widehat{\mathcal{F}}, \mathsf{Enc}(\mathcal{F})) = \mathsf{true}$ and $\mathsf{SlotCheck}(\mathsf{Enc}(\mathcal{F})) = \mathsf{true}$; then add it to G_i.
09: Verify that $\mathrm{BatchVerifyEval}(crs, \hat{S}, \vec{y}_j^S, (j), t; \phi_j^S) = \mathsf{true}$.
10: Verify that $\mathsf{LinkCheck}(\mathsf{Enc}(\mathcal{F}), \{\vec{y}_j^S[k,j]\}_{k\in[L]}, j) = \mathsf{true}$.
11: If all checks pass, set $Z_j \leftarrow \{\vec{y}_j^S[idx, j]\}_{idx\in\mathcal{I}}$ and add (j, Z_j) to R_i.
12: If $|R_i| \geq d_{\mathcal{I}} + 1$ and $rec_i = \mathsf{false}$:
13: Choose $J \subseteq \{j : (j, \cdot) \in R_i\}$ with $|J| = d_{\mathcal{I}} + 1$.
14: For each $j \in J$, compute $\lambda_j \leftarrow \prod_{m\in J\setminus\{j\}} \frac{-m}{j-m}$.
15: For each $idx \in \mathcal{I}$, compute $z_{idx} \leftarrow \sum_{j\in J} \lambda_j Z_j[idx]$.
16: Set $rec_i \leftarrow \mathsf{true}$ and output $(\mathsf{out}, \mathsf{reconstructed}, \{z_{idx}\}_{idx\in\mathcal{I}})$.

Figure 3: The batch reconstruction stage of B1itz for node P_i and tag ID.d.

while $\mathsf{Enc}(\mathcal{F})$ uses a common group base, parties verify their consistency using a pairing-based linear check against the committed bases. Concretely, let $F_k(X) = \sum_{\ell=0}^{2t} f_{k,\ell} X^\ell$, $\widehat{\mathcal{F}}[k] = \prod_{\ell=0}^{2t} g_\ell^{f_{k,\ell}}$, and $\mathsf{Enc}(F_k)[\ell] = g^{f_{k,\ell}}$. The CRS additionally contains paired bases $\{\bar{g}_\ell\}_{\ell=0}^{2t}$ satisfying $e(g_\ell, h) = e(g, \bar{g}_\ell)$. To check all L slots at once, parties derive random coefficients $\alpha_1, \ldots, \alpha_L$ by Fiat–Shamir, set $C_\alpha = \prod_{k=1}^{L} \widehat{\mathcal{F}}[k]^{\alpha_k}$ and $E_{\ell,\alpha} = \prod_{k=1}^{L} \mathsf{Enc}(F_k)[\ell]^{\alpha_k}$, and accept iff:

$$e(C_\alpha, h) = \prod_{\ell=0}^{2t} e(E_{\ell,\alpha}, \bar{g}_\ell).$$

For honestly generated data, the equation follows from bilinearity. If any encoded coefficient vector is inconsistent with the corresponding commitment, the random linear combination passes only with negligible probability. Once this check succeeds, coefficient tests performed on $\mathsf{Enc}(\mathcal{F})$ are sound for the polynomials committed in $\widehat{\mathcal{F}}$.

- SlotCheck. This check verifies the validity of regular slots and high-threshold slots. Index the first N slots as regular and the next B slots as high-threshold. SlotCheck is passed iff:

$$\mathsf{Enc}(F_k)[\ell] = 1_{\mathbb{G}} \quad \text{for all } k \in [N],\ \ell \in \{t+1, \ldots, 2t\},$$
$$\mathsf{Enc}(F_{N+k})[0] = 1_{\mathbb{G}} \quad \text{for all } k \in [B].$$

The first condition enforces that regular slots have a degree of at most t, and the second condition enforces that high-threshold slots share zero.

- LinkCheck. This check links a receiver-specific scalar vector to the committed recovery polynomials. Given $\mathsf{Enc}(\mathcal{F})$, a point x, and a scalar vector $v = (v_1, \ldots, v_L)$ that has already passed $\mathrm{BatchVerifyEval}$ under the corresponding S-batch commitment, LinkCheck accepts iff, for every $k \in [L]$,

$$g^{v_k} = \prod_{\ell=0}^{2t} \mathsf{Enc}(F_k)[\ell]^{x^\ell}.$$

An interesting fact is that we can use our coefficient encoding to simplify the computation by BatchHaven. Namely, since CoeffCheck binds $\mathsf{Enc}(\mathcal{F})$ to $\widehat{\mathcal{F}}$, this proves under DLog binding that the scalar certified by $\mathrm{BatchVerifyEval}$ is exactly $F_k(x)$. This is the concrete check that replaces BatchHaven's extra T-polynomial commitment and evaluation proof: the expected group-encoded value is computed directly from $\mathsf{Enc}(F_k)$, so no separate T batch is needed.

The reconstruction stage. The reconstruction stage is shown in Figure 3. In Lines 01–02, a party initializes the reconstruction buffer for a requested slot set $\mathcal{I}$. In Lines 03–05, it parses its local B1itz output, sets $d_{\mathcal{I}} = t$ for regular-only reconstruction and $d_{\mathcal{I}} = 2t$ otherwise, and sends its local evaluation vector together with the validation data. In Lines 06–11, a receiver repeats the root, cached global validation, batched evaluation, and linkage checks before accepting P_j's reconstruction values for $\mathcal{I}$. In Lines 12–16, after collecting $d_{\mathcal{I}} + 1$ valid points, the party computes one Lagrange basis for the chosen reconstruction set and reuses it to reconstruct every slot in $\mathcal{I}$.

Complexity. B1itz achieves $O(n^2)$ message complexity. All amortized costs in this paragraph are per B1itz slot, where one instance has $L = N + B$ slots. The amortized communication complexity is $O(\lambda n^2)$ and the amortized exponentiation complexity is $O(n)$. The receiver-specific S-polynomial layer is batched across the L slots, and B1itz no longer needs the additional T-polynomial commitment and evaluation route. Instead, each accepted local or reconstruction packet runs LinkCheck, which uses group exponentiations and multiplications over the already published coefficient encodings. The only pairing computation is CoeffCheck. For one L-slot root commitment, the batched equation above costs $2t + 2 = O(n)$ pairings: one pairing on the aggregated commitment and $2t + 1$ pairings against the paired CRS bases. Therefore, for the accepted root commitment in one sharing instance, all n parties perform

$O(n^2)$ pairings in total, i.e., $O(n^2/L)$ amortized pairings per slot. The coefficient encodings add $O(Lt)$ group elements to the send phase. Since a B1itz instance already batches L slots and the encoded coefficients are sent once for the whole batch, $O(t)$ group elements per slot are needed.

Theorem 1 (Security of B1itz)**.** *In the random oracle model, assuming DLog hardness in the bilinear source groups and independently sampled paired CRS bases, B1itz is a secure bHACSS protocol. Specifically, it satisfies* Privacy*,* Liveness*,* Correctness*, and* Completeness*.*

We defer the proof to Appendix A.

6. Asynchronous Complete Random Double Sharing (ACRDS)

In our definition of ACRDS, it generates $N+B$ random secrets, where the first N are t-shared and the last B are *double-shared* (i.e., both t-share and $2t$-shared). The protocol outputs $(N+B)$ t-shares in total, among which the last B also have corresponding $2t$-shares. Together with the shares, ACRDS returns public validation data vdata for the derived output polynomials. This data contains the derived coefficient encodings and the corresponding commitment-consistency information, so any party can locally compute group encodings of output-polynomial evaluations when needed.

The workflow. We show the workflow of our ACRDS protocol Figure 5 and the pseudocode in Figure 4. The protocol runs in four phases: initialization, commitment, agreement, and derivation, as shown below.

- **Initialization.** In Lines 01–03, each party first computes

$$\widetilde{N} \leftarrow \lceil N/(t+1) \rceil, \qquad \widetilde{B} \leftarrow \lceil B/(t+1) \rceil,$$

  sets $\widehat{N} \leftarrow \widetilde{N} + \widetilde{B}$ and $\widehat{B} \leftarrow \widetilde{B}$, and initializes the local dealer set, the agreed set, the validated-share buffer, and vdata.
- **Commitment.** In Lines 04–05, every party P_i samples a vector of $\widehat{N}$ field elements $(s_1^{(i)}, \ldots, s_{\widehat{N}}^{(i)})$ and queries a B1itz instance $\mathsf{B1itz}_i$ to share $(s_1^{(i)}, \ldots, s_{\widehat{N}}^{(i)})$ in the regular slots, and zeros for the $\widehat{B}$ high-threshold slots. Note that the coefficient encodings in B1itz are used as the validation data, so if one B1itz instance is completed by any honest party, all honest parties eventually hold the validation data.
  Meanwhile, P_i also participates in other B1itz instances. In Lines 06–10, upon completing the sharing stage of any $\mathsf{B1itz}_j$, P_i adds j, the index of the dealer P_j, to a local buffer S_i. Furthermore, in the sharing stage of B1itz, each party P_i receives $\widehat{N}$ degree-t secret shares $\{[\![s_m^{(j)}]\!]_t\}_{m=1}^{\widehat{N}}$ and $\widehat{B}$ degree-$2t$ zero shares $\{[\![0_k^{(j)}]\!]_{2t}\}_{k=1}^{\widehat{B}}$. It can then directly compute

$$[\![\widetilde{s}_k^{(j)}]\!]_{2t} \leftarrow [\![s_{\widetilde{N}+k}^{(j)}]\!]_t + [\![0_k^{(j)}]\!]_{2t}$$

  for each $k \in [1, \widetilde{B}]$, to obtain $\widetilde{B}$ safe degree-$2t$ shares corresponding to the same secrets as the last $\widetilde{B}$ degree-t sharings.
- **Agreement.** In Lines 11–13, once $|S_i| = 2t+1$, P_i invokes MVBA using the n-bit buffer S_i as input. S_i denotes a common subset of dealers, e.g., $CS = \{P_1, \ldots, P_{2t+1}\}$.

▷ **Asynchronous Complete Random Double Sharing (Code for P_i)**
Input: N – number of t-shared secrets to generate; B – number of double-shared secrets to generate.
Output: $\big(\{[\![r_k]\!]_t\}_{k=1}^{N+B},\ \{[\![r_k]\!]_{2t}\}_{k=N+1}^{N+B}, \mathsf{vdata}\big)$.
▷ **Initialization**.
01: $\widetilde{N} \leftarrow \lceil N/(t+1) \rceil$, $\widetilde{B} \leftarrow \lceil B/(t+1) \rceil$, $\widehat{N} \leftarrow \widetilde{N} + \widetilde{B}$, and $\widehat{B} \leftarrow \widetilde{B}$.
02: $S_i \leftarrow \emptyset$ be the set of nodes from which P_i has completed the sharing stage.
03: CS be the common subset, $shares \leftarrow \emptyset$ be the validated shares received from other nodes, and vdata $\leftarrow \emptyset$.
▷ **The commitment phase**.
04: Select random secrets $(s_1^{(i)}, \ldots, s_{\widehat{N}}^{(i)})$ and set $\mathbf{s}^{(i)} \leftarrow (s_1^{(i)}, \ldots, s_{\widehat{N}}^{(i)})$.
05: Query the $\mathsf{B1itz}_i$.Share$(\mathsf{in}, \mathsf{share}, \mathbf{s}^{(i)}; \widehat{N}, \widehat{B})$ for $\mathsf{B1itz}_i$ and participate in other B1itz instances.
06: **Upon** completing the sharing stage of $\mathsf{B1itz}_j$:
07: $S_i \leftarrow S_i \cup \{j\}$, where j is the index of $\mathsf{B1itz}_j$.
08: **For** $k = 1$ to $\widetilde{B}$
09: Compute $[\![\widetilde{s}_k^{(j)}]\!]_{2t} \leftarrow [\![s_{\widetilde{N}+k}^{(j)}]\!]_t + [\![0_k^{(j)}]\!]_{2t}$.
10: $shares \leftarrow shares \cup \big\{\{[\![s_m^{(j)}]\!]_t\}_{m=1}^{\widehat{N}},\ \{[\![\widetilde{s}_k^{(j)}]\!]_{2t}\}_{k=1}^{\widetilde{B}}\big\}$.
▷ **The agreement phase**.
11: **if** $|S_i| = 2t+1$ **then**
12: Call MVBA using S_i as input.
13: Wait for MVBA output S, let $CS \leftarrow S$. Order the parties in CS, and let $CS = \{P_1, \ldots, P_{2t+1}\}$.
▷ **The derivation phase**.
14: **For** $k = 1$ to $\widetilde{N}$
15: Compute

$$([\![r_{k,1}]\!]_t, \ldots, [\![r_{k,t+1}]\!]_t)^T \leftarrow M \cdot ([\![s_k^{(1)}]\!]_t, \ldots, [\![s_k^{(2t+1)}]\!]_t)^T.$$

16: **For** $k = 1$ to $\widetilde{B}$
17: Compute

$$([\![r'_{k,1}]\!]_t, \ldots, [\![r'_{k,t+1}]\!]_t)^T \leftarrow M \cdot ([\![s_{\widetilde{N}+k}^{(1)}]\!]_t, \ldots, [\![s_{\widetilde{N}+k}^{(2t+1)}]\!]_t)^T.$$

18: Compute

$$([\![r'_{k,1}]\!]_{2t}, \ldots, [\![r'_{k,t+1}]\!]_{2t})^T \leftarrow M \cdot ([\![\widetilde{s}_k^{(1)}]\!]_{2t}, \ldots, [\![\widetilde{s}_k^{(2t+1)}]\!]_{2t})^T.$$

19: Derive vdata by applying the same linear combinations M to the validated coefficient encodings and commitments of the dealers in CS.
20: Re-index $\{[\![r_{k,\ell}]\!]_t\}_{k\in[\widetilde{N}],\,\ell\in[t+1]}$ as $\{[\![r_k]\!]_t\}_{k=1}^{N}$, and re-index $\{([\![r'_{k,\ell}]\!]_t, [\![r'_{k,\ell}]\!]_{2t})\}_{k\in[\widetilde{B}],\,\ell\in[t+1]}$ as $\{([\![r_k]\!]_t, [\![r_k]\!]_{2t})\}_{k=N+1}^{N+B}$.
21: **Return** $\big(\{[\![r_k]\!]_t\}_{k=1}^{N+B},\ \{[\![r_k]\!]_{2t}\}_{k=N+1}^{N+B}, \mathsf{vdata}\big)$.

Figure 4: ACRDS for node P_i. Here $M \in \mathbb{F}^{(t+1)\times(2t+1)}$ is a public Vandermonde matrix over distinct nonzero points. The protocol internally pads the number of source sharings to $\widetilde{N} = \lceil N/(t+1) \rceil$ and $\widetilde{B} = \lceil B/(t+1) \rceil$, sets $\widehat{N} = \widetilde{N} + \widetilde{B}$ and $\widehat{B} = \widetilde{B}$, and discards the extra derived sharings introduced by padding. The predicate of MVBA $Q(S)$ by each party P_j is set as: for every $j \in S$, $j \in S_i$.

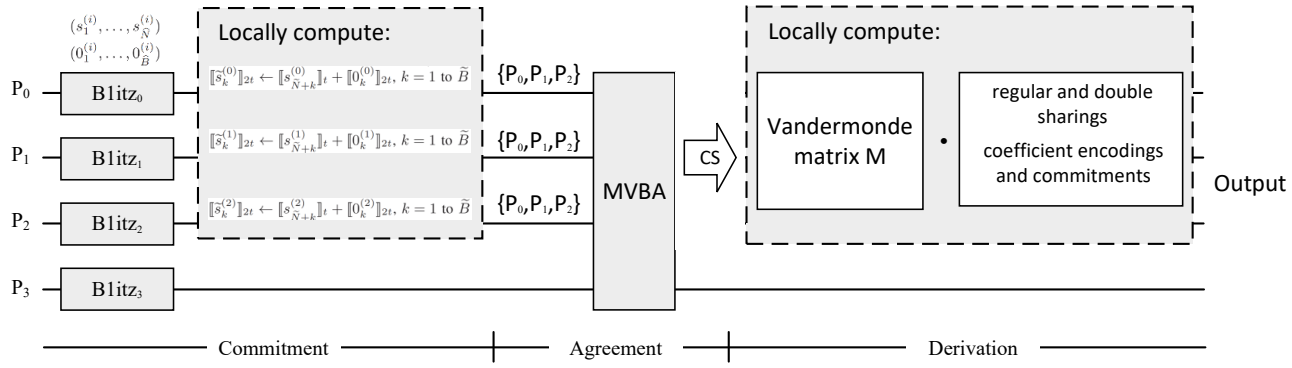


Figure 5: Illustration of the ACRDS protocol.

The predicate of MVBA ensures that for each $j \in S$ where S is the output of MVBA, every party eventually completes the sharing stage of $\mathsf{B1itz}_j$.

- **Derivation.** In Lines 14–21, the derivation phase uses a public Vandermonde matrix over distinct nonzero points

$$M \in \mathbb{F}^{(t+1)\times(2t+1)}$$

to extract multiple output sharings from the sharings contributed by the dealers in CS.
In Lines 14–15, for each $k \in \{1, \ldots, \widetilde{N}\}$, let

$$(\llbracket s_k^{(1)} \rrbracket_t, \ldots, \llbracket s_k^{(2t+1)} \rrbracket_t)^T$$

be the degree-t shares received from the dealers in CS. Party P_i multiplies this vector by M to derive $t+1$ fresh degree-t sharings.
In Lines 16–18, for each $k \in \{1, \ldots, \widetilde{B}\}$, let

$$(\llbracket s_{\widetilde{N}+k}^{(1)} \rrbracket_t, \ldots, \llbracket s_{\widetilde{N}+k}^{(2t+1)} \rrbracket_t)^T$$

and

$$(\llbracket \widetilde{s}_k^{(1)} \rrbracket_{2t}, \ldots, \llbracket \widetilde{s}_k^{(2t+1)} \rrbracket_{2t})^T$$

be the corresponding degree-t and degree-$2t$ shares. Party P_i applies the same matrix M to derive $t+1$ fresh double sharings.
In Line 19, the validation data is derived homomorphically from the same matrix multiplication. Write $F_m^{(j)}$ for the regular polynomial of dealer P_j, $Z_k^{(j)}$ for its k-th high-threshold zero polynomial, $E_m^{(j)} = \mathsf{Enc}(F_m^{(j)})$, and $H_k^{(j)} = \mathsf{Enc}(Z_k^{(j)})$. For group elements or coefficient encodings, define

$$\mathsf{Lin}_\ell(U^{(1)}, \ldots, U^{(2t+1)}) := \prod_{j=1}^{2t+1} (U^{(j)})^{M[\ell,j]}.$$

When each $U^{(j)}$ is a coefficient-encoding vector, the same product is applied independently to each coefficient position. Thus the scalar shares, commitments, and coefficient encodings are all derived using the same row of M; only the underlying operation changes from field addition to group multiplication. For a derived regular output $R_{k,\ell} = \sum_j M[\ell, j] F_k^{(j)}$, vdata stores

$$\widehat{R}_{k,\ell} = \mathsf{Lin}_\ell(\widehat{F}_k^{(1)}, \ldots, \widehat{F}_k^{(2t+1)}),$$
$$\mathsf{Enc}(R_{k,\ell}) = \mathsf{Lin}_\ell(E_k^{(1)}, \ldots, E_k^{(2t+1)}).$$

For a derived double output, vdata stores the same data for $R'_{k,\ell} = \sum_j M[\ell, j] F_{\widetilde{N}+k}^{(j)}$ and for $\widetilde{R}_{k,\ell} = \sum_j M[\ell, j](F_{\widetilde{N}+k}^{(j)} + Z_k^{(j)})$:

$$\widehat{R}'_{k,\ell} = \mathsf{Lin}_\ell(\widehat{F}_{\widetilde{N}+k}^{(1)}, \ldots, \widehat{F}_{\widetilde{N}+k}^{(2t+1)}),$$
$$\mathsf{Enc}(R'_{k,\ell}) = \mathsf{Lin}_\ell(E_{\widetilde{N}+k}^{(1)}, \ldots, E_{\widetilde{N}+k}^{(2t+1)}),$$
$$\widehat{\widetilde{R}}_{k,\ell} = \mathsf{Lin}_\ell(\widehat{F}_{\widetilde{N}+k}^{(1)} \widehat{Z}_k^{(1)}, \ldots, \widehat{F}_{\widetilde{N}+k}^{(2t+1)} \widehat{Z}_k^{(2t+1)}),$$
$$\mathsf{Enc}(\widetilde{R}_{k,\ell}) = \mathsf{Lin}_\ell(E_{\widetilde{N}+k}^{(1)} H_k^{(1)}, \ldots, E_{\widetilde{N}+k}^{(2t+1)} H_k^{(2t+1)}).$$

Any party can verify the derived commitment/encoding consistency with CoeffCheck and locally compute $g^{R(x)}$ from $\mathsf{Enc}(R)$ at any public point x. This step is local and introduces no extra communication. In Lines 20–21, the derived sharings are re-indexed, and the first N degree-t sharings together with the first B double sharings and vdata are returned as the output of $\mathsf{ACRDS}(N, B)$.

Correctness. We show the correctness of our ACRDS protocol in Appendix B.

7. TRIO: The Triple Generation Protocol

Overview. TRIO generates Beaver triples directly from one ACRDS invocation. For D triples, the parties call $\mathsf{ACRDS}(2D, D)$ to obtain D degree-t sharings $\{\llbracket a_k \rrbracket_t\}_{k\in[D]}$, another D degree-t sharings $\{\llbracket b_k \rrbracket_t\}_{k\in[D]}$, and D double sharings $\{(\llbracket r_k \rrbracket_t, \llbracket r_k \rrbracket_{2t})\}_{k\in[D]}$, together with public validation data vdata. Each party locally computes the degree-$2t$ masked product share $p_k^{(i)} = \llbracket a_k \rrbracket_t^{(i)} \llbracket b_k \rrbracket_t^{(i)} -$

▷ **TRIO (Triples Output protocol) – Code for** P_i
Input: system parameters (n, t), the amount of Beaver triples D.
Output: $\{(\llbracket a_j \rrbracket_t^{(i)}, \llbracket b_j \rrbracket_t^{(i)}, \llbracket c_j \rrbracket_t^{(i)})\}_{j=1}^{D}$.
▷ **Generate random sharings.**
01: $\mathsf{triples} \leftarrow \emptyset$, $Q_i \leftarrow \emptyset$.
02: Invoke $\mathsf{ACRDS}(2D, D)$ and parse the output as $\{\llbracket a_k \rrbracket_t^{(i)}\}_{k=1}^{D}$, $\{\llbracket b_k \rrbracket_t^{(i)}\}_{k=1}^{D}$, and $\{(\llbracket r_k \rrbracket_t^{(i)}, \llbracket r_k \rrbracket_{2t}^{(i)})\}_{k=1}^{D}$, together with vdata.
▷ **Mask local products.**
03: **For** $k = 1$ to D:
04: $\quad p_k^{(i)} \leftarrow \llbracket a_k \rrbracket_t^{(i)} \cdot \llbracket b_k \rrbracket_t^{(i)} - \llbracket r_k \rrbracket_{2t}^{(i)}$.
05: Broadcast $(\mathsf{open}, i, \mathbf{p}_i, \mathsf{vdata})$ where $\mathbf{p}_i = (p_1^{(i)}, \ldots, p_D^{(i)})$.
▷ **Verified batch opening.**
06: **Upon receiving** $(\mathsf{open}, j, \mathbf{p}_j, \mathsf{vdata}_j)$ from P_j:
07: $\quad$ Verify $\mathsf{vdata}_j = \mathsf{vdata}$ and $\mathsf{CoeffCheck}$ for vdata_j.
08: $\quad$ For each $k \in [D]$, compute from the verified encodings in vdata_j $A_{k,j} = g^{a_k(j)}$, $B_{k,j} = g^{b_k(j)}$, and $R_{k,j} = g^{\widetilde{r}_k(j)}$.
09: $\quad$ Sample $\gamma_1, \ldots, \gamma_D \xleftarrow{\$} \mathbb{F}$ and check

$$\prod_{k=1}^{D} e(A_{k,j}^{\gamma_k}, B_{k,j}) = e\left(\prod_{k=1}^{D} (g^{p_k^{(j)}})^{\gamma_k}, g\right) \cdot e\left(\prod_{k=1}^{D} R_{k,j}^{\gamma_k}, g\right),$$

$\quad$ and if it holds, add j to Q_i.
10: If $|Q_i| \geq 2t + 1$, choose $J \subseteq Q_i$ with $|J| = 2t + 1$ and precompute $\{\lambda_j\}_{j \in J}$ for interpolation at 0.
11: **For** $k = 1$ to D:
12: $\quad d_k \leftarrow \sum_{j \in J} \lambda_j p_k^{(j)}$ and $\llbracket c_k \rrbracket_t^{(i)} \leftarrow d_k + \llbracket r_k \rrbracket_t^{(i)}$.
13: $\quad \mathsf{triples} \leftarrow \mathsf{triples} \cup \{(\llbracket a_k \rrbracket_t^{(i)}, \llbracket b_k \rrbracket_t^{(i)}, \llbracket c_k \rrbracket_t^{(i)})\}$.
14: **Return** $\mathsf{triples}$.

Figure 6: TRIO generates D Beaver triples per invocation for node P_i.

$\llbracket r_k \rrbracket_{2t}^{(i)}$, batch-opens the vector $\mathbf{p}_i$, reconstructs $d_k = a_k b_k - r_k$, and finally sets $\llbracket c_k \rrbracket_t^{(i)} = d_k + \llbracket r_k \rrbracket_t^{(i)}$.

The key benefit is that TRIO avoids a separate re-sharing protocol in the degree-reduction step. The double sharing from ACRDS already provides the degree-$2t$ mask. The opening share of $a_k b_k - r_k$ can be verified where the proof is derived from the public coefficient encodings in vdata. Moreover, all D openings use the same reconstruction set, so the Lagrange coefficients are computed once and reused across the whole batch. Finally, a random linear combination compresses the D product-verification equations for one sender into a single batched pairing equation, which substantially reduces verification overhead compared with checking the D products independently.

The workflow. We show the pseudocode of TRIO in Figure 6. The protocol consists of three steps.

- **Random sharings.** In Lines 01–02, the parties initialize local state and invoke $\mathsf{ACRDS}(2D, D)$. The first D degree-t outputs are parsed as $\{a_k\}_{k \in [D]}$, the next D degree-t outputs are parsed as $\{b_k\}_{k \in [D]}$, and the D double outputs are parsed as $\{r_k\}_{k \in [D]}$. For each double output r_k, let $\widetilde{r}_k(\cdot)$ denote the degree-$2t$ polynomial underlying $\llbracket r_k \rrbracket_{2t}$. The vdata contains the coefficient encodings and commitments for these output polynomials. In the opening message we write vdata explicitly to bind the opening to this ACRDS output. This can be optimized by sending a digest or an identifier without affecting the correctness of the protocol.
- **Masked products.** In Lines 03–05, for each $k \in [D]$, party P_i locally computes $p_k^{(i)} = \llbracket a_k \rrbracket_t^{(i)} \llbracket b_k \rrbracket_t^{(i)} - \llbracket r_k \rrbracket_{2t}^{(i)}$ and sends the vector $\mathbf{p}_i = (p_1^{(i)}, \ldots, p_D^{(i)})$ together with vdata. Since the product of two degree-t shares and the mask $\llbracket r_k \rrbracket_{2t}$ both have a threshold of $2t$, the vector $\mathbf{p}_i$ is a batch of local evaluations of the degree-$2t$ polynomials $a_k b_k - r_k$.
- **Verified opening and output.** In Lines 06–09, upon receiving $(\mathsf{open}, j, \mathbf{p}_j, \mathsf{vdata}_j)$, a party first checks that vdata_j matches the ACRDS output and passes $\mathsf{CoeffCheck}$. It then derives $A_{k,j} = g^{a_k(j)}$, $B_{k,j} = g^{b_k(j)}$, and $R_{k,j} = g^{\widetilde{r}_k(j)}$ from the encoded polynomials in vdata_j. For random coefficients $\gamma_1, \ldots, \gamma_D$, the receiver accepts P_j's opening vector only if:

$$\prod_{k=1}^{D} e(A_{k,j}^{\gamma_k}, B_{k,j}) = e\left(\prod_{k=1}^{D} (g^{p_k^{(j)}})^{\gamma_k}, g\right) \cdot e\left(\prod_{k=1}^{D} R_{k,j}^{\gamma_k}, g\right).$$

In Lines 10–14, after collecting $2t + 1$ accepted vectors, the party reconstructs all $d_k = a_k b_k - r_k$ using one precomputed interpolation basis and outputs $\llbracket c_k \rrbracket_t^{(i)} = d_k + \llbracket r_k \rrbracket_t^{(i)}$.

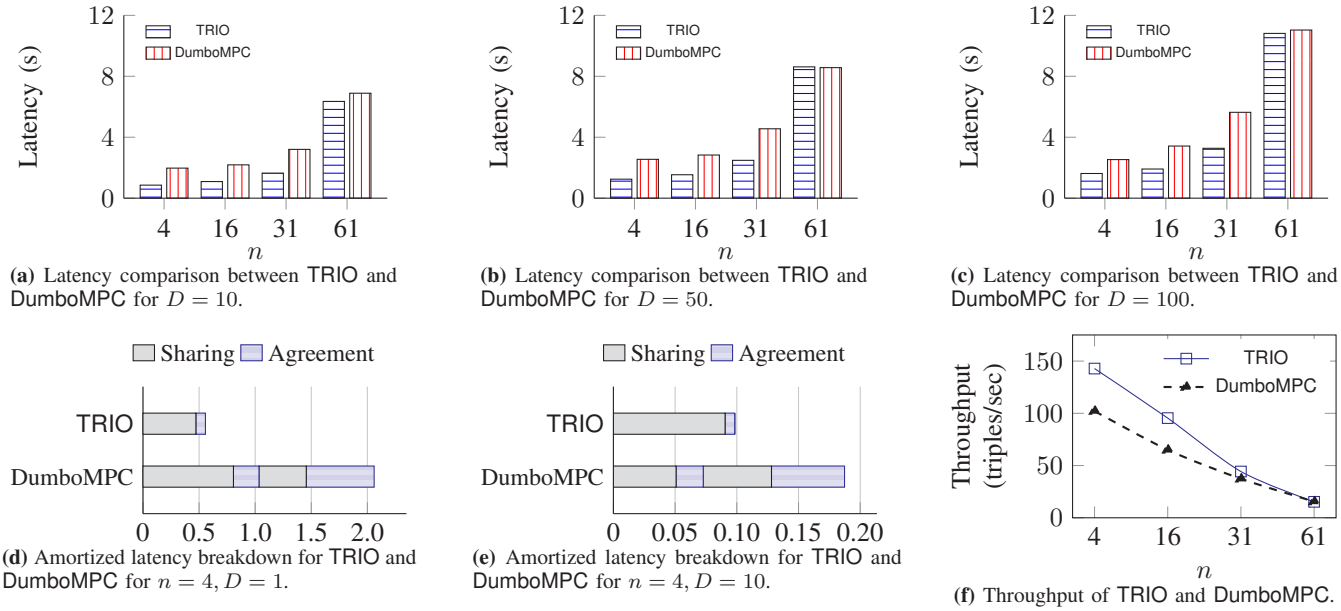
Discussion. Based on TRIO, we can generate Beaver triples with G.O.D. guarantee. We can then apply the standard workflow for the online phase and build our RondoMPC protocol. We show the proof of TRIO in Appendix C.

8. Implementation and Evaluation

We implement RondoMPC in Golang, and our evaluation focuses on the Beaver triple generation, i.e., our TRIO protocol. We use the Kyber crypto library[1] to implement our batched polynomial commitment scheme and the pairing-based checks used by coefficient encodings. In the same library, we also implement BatchHaven [47] as the baseline to compare the additional cost introduced by B1itz. We also compare the performance of DumboMPC using their open-source implementation[2]. We use gRPC as the communication library. We use HMAC to realize the authenticated channel and SHA256 as the hash function. Our implementation involves around 26,000 new LOC for the system and about 1,500 LOC for evaluation.

We evaluate the performance of our protocols on Amazon EC2 using up to 61 virtual machines (VMs). We use *c5.4xlarge* instances (16 virtual CPUs and 32GB memory) for our evaluation. We distribute the nodes evenly in four different regions: us-west-2 (Oregon, US), us-east-2 (Ohio, US), ap-southeast-1 (Singapore), and eu-west-1 (Ireland). We vary the network size n and batch size D in our evaluation. We use n to denote the network size, where $n = 3t+1$ nodes are launched in each experiment. Unless otherwise stated, we report the performance for $n \in \{4, 16, 31, 61\}$

1. The Kyber crypto library: https://github.com/dedis/kyber
2. DumboMPC implementation: https://github.com/dcy456/Dumbo-MPC.



(a) Latency comparison between TRIO and DumboMPC for $D = 10$.

(b) Latency comparison between TRIO and DumboMPC for $D = 50$.

(c) Latency comparison between TRIO and DumboMPC for $D = 100$.

(d) Amortized latency breakdown for TRIO and DumboMPC for $n = 4, D = 1$.

(e) Amortized latency breakdown for TRIO and DumboMPC for $n = 4, D = 10$.

(f) Throughput of TRIO and DumboMPC.

Figure 7: Evaluation results for TRIO. The breakdown plots report the amortized latency per triple by dividing each sharing/agreement stage by the number of generated triples.

and $D \in \{10, 50, 100, 200, 500\}$. We repeat each experiment five times and report the average results.

For the comparison with DumboMPC, we focus on the *slow path* (that extends GS23), as the fast path is not AMPC with G.O.D. We do not compare the performance with other practical AMPC protocols, such as Velox, as Velox does not achieve G.O.D.

Latency comparison. Figure 7a-7c compare the end-to-end latency of TRIO with DumboMPC for $D \in \{10, 50, 100\}$. As shown in the figure, the measurements confirm the round-efficiency advantage of TRIO: for $n \leq 31$, TRIO consistently has lower latency and is $1.56\times$–$2.31\times$ faster than DumboMPC. This is expected, as RondoMPC only has one phase of secret sharing and consensus. At $n = 61$, the performance of the protocols becomes close. We believe this is because the wide-area communication in the WAN becomes the bottleneck.

Latency breakdown. We next compare the amortized latency breakdown of TRIO and DumboMPC in Figure 7d-7e. We focus on the sharing/agreement phases, mainly because the reconstruction is negligible in both protocols. For TRIO, the sharing and agreement correspond to ACRDS and MVBA, respectively. For DumboMPC, the same applies to the share and re-share phases. Our experimental results show that the design of only one-phase consensus in TRIO can effectively lower the latency. The latency improvement of TRIO is particularly notable for smaller D. For $D = 1$ and 10, the latency of TRIO is $1.91\times - 3.7\times$ lower than that of DumboMPC.

Throughput and scalability. We report the throughput as the number of Beaver triples generated per second for each network size in Figure 7f. The best throughput is obtained at $D = 500$ for both protocols. For $n \leq 31$, TRIO consistently outperforms DumboMPC, where the throughput is 18.6%–46.8% higher. The results match our latency evaluation. However, the performance gain of RondoMPC is not as large as that for our latency evaluation. Namely, the throughput is higher with a larger batch size. However, with a larger D, the computation cost is much higher. Thus, although we only have one phase of secret sharing and consensus, the computational cost dominates the performance. Thus, the throughput improvement is not that significant for a larger D. When $n = 61$, the two protocols are nearly identical, where the throughput of TRIO is 1.9% higher than that of DumboMPC.

9. Conclusion

We present RondoMPC, an asynchronous multiparty computation (AMPC) protocol with guaranteed output delivery (G.O.D.). Our protocol focuses on Beaver triple generation, a well known offline phase that can entirely removes the communication cost for online computation. Unlike prior practical AMPC constructions with G.O.D. that requires two phases of consensus and secret sharing, our Beaver triple generation protocol TRIO only requires one phase. Our experimentation results show that compared to the state-of-the-art AMPC protocol, our protocol has up to 46.8% higher throughput for Beaver triple generation, and the latency of generating each triple is up to $3.70\times$ lower.

References

[1] M. Ben-Or, S. Goldwasser, and A. Wigderson, "Completeness theorems for non-cryptographic fault-tolerant distributed computation," in

Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali, 2019, pp. 351–371.

[2] D. Chaum, C. Crépeau, and I. Damgard, "Multiparty unconditionally secure protocols," in *STOC*, 1988, pp. 11–19.

[3] A. C. Yao, "Theory and application of trapdoor functions," in *23rd Annual Symposium on Foundations of Computer Science (SFCS 1982)*. IEEE, 1982, pp. 80–91.

[4] O. Goldreich, S. Micali, and A. Wigderson, "How to play any mental game, or a completeness theorem for protocols with honest majority," in *Providing sound foundations for cryptography: on the work of Shafi Goldwasser and Silvio Micali*, 2019, pp. 307–328.

[5] G. Asharov and Y. Lindell, "A full proof of the bgw protocol for perfectly secure multiparty computation," *Journal of Cryptology*, vol. 30, no. 1, pp. 58–151, 2017.

[6] D. Lu, T. Yurek, S. Kulshreshtha, R. Govind, A. Kate, and A. Miller, "HoneybadgerMPC and asynchromix: Practical asynchronous MPC and its application to anonymous communication," in *CCS*, 2019, pp. 887–903.

[7] I. Abraham, B. Pinkas, and A. Yanai, "Blinder–scalable, robust anonymous committed broadcast," in *CCS*, 2020, pp. 1233–1252.

[8] F. Benhamouda, S. Halevi, and T. Halevi, "Supporting private data on hyperledger fabric with secure multiparty computation," *IBM Journal of Research and Development*, vol. 63, no. 2/3, pp. 3–1, 2019.

[9] D. Maram, H. Malvai, F. Zhang, N. Jean-Louis, A. Frolov, T. Kell, T. Lobban, C. Moy, A. Juels, and A. Miller, "Candid: Can-do decentralized identity with legacy compatibility, sybil-resistance, and accountability," in *S&P*. IEEE, 2021, pp. 1348–1366.

[10] Y. Li, Y. Duan, Z. Huang, C. Hong, C. Zhang, and Y. Song, "Efficient 3PC for binary circuits with application to Maliciously-Secure DNN inference," in *32nd USENIX Security Symposium*, 2023, pp. 5377–5394.

[11] E. Ben-Sasson, A. Chiesa, M. Green, E. Tromer, and M. Virza, "Secure sampling of public parameters for succinct zero knowledge proofs," in *S&P*. IEEE, 2015, pp. 287–304.

[12] M. Ben-Or, B. Kelmer, and T. Rabin, "Asynchronous secure computations with optimal resilience," in *PODC*. ACM, 1994, pp. 183–192.

[13] A. Patra, A. Choudhary, and C. P. Rangan, "Efficient statistical asynchronous verifiable secret sharing with optimal resilience," in *International Conference on Information Theoretic Security*. Springer, 2009, pp. 74–92.

[14] A. Choudhury and A. Patra, "On the communication efficiency of statistically secure asynchronous mpc with optimal resilience," *Journal of Cryptology*, vol. 36, no. 2, p. 13, 2023.

[15] A. Bandarupalli, X. Ji, A. Kate, C.-D. Liu-Zhang, and Y. Song, "Computationally efficient asynchronous mpc with linear communication and low additive overhead," in *Annual International Cryptology Conference*. Springer, 2025, pp. 261–294.

[16] V. Goyal, C.-D. Liu-Zhang, and Y. Song, "Towards achieving asynchronous mpc with linear communication and optimal resilience," in *Annual International Cryptology Conference*. Springer, 2024, pp. 170–206.

[17] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen, "Asynchronous multiparty computation: Theory and implementation," in *International workshop on public key cryptography*. Springer, 2009, pp. 160–179.

[18] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 1–18.

[19] M. Keller, "Mp-spdz: A versatile framework for multi-party computation," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 1575–1590.

[20] A. Bandarupalli, X. Ji, A. Kate, C.-D. Liu-Zhang, D. Pöllmann, and Y. Song, "Velox: Scalable fair asynchronous mpc from lightweight cryptography," in *CCS*, 2025, pp. 1799–1813.

[21] J. Groth and V. Shoup, "Design and analysis of a distributed ecdsa signing service," *Cryptology ePrint Archive*, 2022.

[22] Y. Su, Y. Lu, J. Li, Y. Wang, C. Dong, and Q. Tang, "Dumbo-mpc: Efficient fully asynchronous mpc with optimal resilience," in *Usenix Security*, 2025.

[23] W. Yu, M. Xu, B. Wu, S. Duan, and X. Cheng, "AD-MPC: Asynchronous dynamic MPC with guaranteed output delivery," in *CCS*, 2025.

[24] D. Beaver, "Efficient multiparty protocols using circuit randomization," in *Annual international cryptology conference*. Springer, 1991, pp. 420–432.

[25] I. Damgård and J. B. Nielsen, "Scalable and unconditionally secure multiparty computation," in *Annual International Cryptology Conference*. Springer, 2007, pp. 572–590.

[26] Z. Beerliová-Trubíniová and M. Hirt, "Perfectly-secure mpc with linear communication complexity," in *Theory of Cryptography Conference*. Springer, 2008, pp. 213–230.

[27] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *International conference on the theory and application of cryptology and information security*. Springer, 2010, pp. 177–194.

[28] S. Duan, X. Wang, and H. Zhang, "Fin: Practical signature-free asynchronous common subset in constant time," in *CCS*, 2023, pp. 815–829.

[29] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song, "The honey badger of bft protocols," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 31–42.

[30] S. Das, S. Duan, S. Liu, A. Momose, L. Ren, and V. Shoup, "Asynchronous consensus without trusted setup or public-key cryptography," in *CCS*, 2024, pp. 3242–3256.

[31] B. Guo, Y. Lu, Z. Lu, Q. Tang, J. Xu, and Z. Zhang, "Speeding dumbo: Pushing asynchronous bft closer to practice," *NDSS*, 2022.

[32] T. Yurek, L. Luo, J. Fairoze, A. Kate, and A. Miller, "hbacss: How to robustly share many secrets," in *NDSS*, 2022.

[33] S. Das, Z. Xiang, A. Tomescu, A. Spiegelman, B. Pinkas, and L. Ren, "Verifiable secret sharing simplified," in *S&P*. IEEE, 2025, pp. 633–651.

[34] A. Momose, S. Das, and L. Ren, "On the security of kzg commitment for vss," in *CCS*, 2023, pp. 2561–2575.

[35] A. Momose, "Practical asynchronous mpc from lightweight cryptography," *Cryptology ePrint Archive*, 2024.

[36] V. Goyal, H. Li, R. Ostrovsky, A. Polychroniadou, and Y. Song, "Atlas: efficient and scalable mpc in the honest majority setting," in *Annual International Cryptology Conference*. Springer, 2021, pp. 244–274.

[37] J. Groth and V. Shoup, "Design and analysis of a distributed ecdsa signing service," *Cryptology ePrint Archive*, 2022.

[38] D. Beaver, S. Micali, and P. Rogaway, "The round complexity of secure protocols," in *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, 1990, pp. 503–513.

[39] Z. Beerliová-Trubíniová and M. Hirt, "Perfectly-secure mpc with linear communication complexity," in *Theory of Cryptography Conference*. Springer, 2008, pp. 213–230.

[40] S. Das, Z. Xiang, L. Kokoris-Kogias, and L. Ren, "Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling," in *USENIX Security*, 2023, pp. 5359–5376.

[41] M. Ben-Or, R. Canetti, and O. Goldreich, "Asynchronous secure computation," in *STOC*. ACM, 1993, pp. 52–61.

[42] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup, "Secure and efficient asynchronous broadcast protocols," in *CRYPTO*. Springer, 2001, pp. 524–541.

[43] I. Abraham, D. Malkhi, and A. Spiegelman, "Asymptotically optimal validated asynchronous byzantine agreement," in *PODC*. ACM, 2019, pp. 337–346.

[44] Y. Lu, Z. Lu, Q. Tang, and G. Wang, "Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited," in *PODC*, 2020.

[45] S. Das, T. Yurek, Z. Xiang, A. K. Miller, L. Kokoris-Kogias, and L. Ren, "Practical asynchronous distributed key generation," in *IEEE Symposium on Security and Privacy*. IEEE, 2022, pp. 2518–2534.

[46] T. P. Pedersen, "Non-interactive and information-theoretic secure verifiable secret sharing," in *Annual international cryptology conference*. Springer, 1991, pp. 129–140.

[47] X. Meng, X. Sui, Z. Yang, K. Rong, W. Xu, S. Chen, Y. Yan, and S. Duan, "Rondo: Scalable and reconfiguration-friendly randomness beacon," in *NDSS*, 2025.

[48] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *S&P*. IEEE, 2018, pp. 315–334.

[49] G. Bracha, "Asynchronous byzantine agreement protocols," *Information and Computation*, vol. 75, no. 2, pp. 130–143, 1987.

Appendix A. Appendix: B1itz Details and Security Proof

Batched evaluation proof. B1itz uses the same $\mathrm{BatchEval}$ and $\mathrm{BatchVerifyEval}$ procedures as BatchHaven [47]; only notation changes: the committed batch is $\mathcal{F} = \{F_1, \ldots, F_L\}$, its commitment is $\widehat{\mathcal{F}}$, the stacked coefficient matrix is $\mathbf{s}$, and the evaluation selector is $\mathbf{y}$. We use the procedures as a black box with two properties. First, $\mathrm{BatchEval}$ produces evaluations Y and proofs Φ showing that $Y = \mathbf{s}\mathbf{y}$ is consistent with $\widehat{\mathcal{F}}$. Second, except with negligible probability, $\mathrm{BatchVerifyEval}(crs, \widehat{\mathcal{F}}, v, x, p; \phi) = \mathsf{true}$ implies that v is the evaluation at x of the polynomial batch committed by $\widehat{\mathcal{F}}$. The Bulletproof-style inner-product proof is Fiat–Shamir transformed, batched across evaluation points, and simulatable in the random-oracle model, as in BatchHaven.

A.1. Complexity Accounting

The batched evaluation proofs use the same exponentiation cost as in BatchHaven. All amortized costs in this paragraph are per B1itz slot, where one instance has $L = N + B$ slots. The additional pairing work in B1itz comes from the pairing-based CoeffCheck, which is run once per accepted root commitment after the static validation data has been cached. Hence one L-slot B1itz instance uses $O(n^2)$ pairings for coefficient consistency, or $O(n^2/L)$ amortized pairings per slot. This is the pairing cost used by the ACRDS and TRIO analyses.

A.2. Security Proof of B1itz

Theorem 1. *In the random oracle model, assuming DLog hardness in the bilinear source groups and independently sampled paired CRS bases,* B1itz *is a secure bHACSS protocol. Specifically, it satisfies* Privacy*,* Liveness*,* Correctness*, and* Completeness*.*

We prove the theorem. We use the following two properties of the batched evaluation proof inherited from BatchHaven.

- *Evaluation binding.* Except with negligible probability, if $\mathrm{BatchVerifyEval}(crs, \widehat{\mathcal{F}}, v, x, d; \phi) = \mathsf{true}$, then v is the evaluation at x of the polynomial batch committed by $\widehat{\mathcal{F}}$.
- *Proof simulatability.* In the random-oracle model, the proof components produced by $\mathrm{BatchEval}$ for an accepting statement are simulatable without the polynomial witnesses.

In the sharing stage, the dealer creates degree-t polynomials $S_{k,j}$ such that $S_{k,j}(j) = F_k(j)$. A valid dealer packet for P_i contains evaluations of all $S_{k,j}$ at point i, a batched proof for these S-evaluations, and coefficient encodings that allow P_i to check that the delivered local scalar shares equal the evaluations of the committed recovery polynomials. A valid echo message from P_j to P_i contains verified evaluations of the polynomials $S_{k,i}$ at point j.

Let $C \subseteq [n]$ be the corrupted set, with $|C| \le t$, and let $L = N + B$ be the number of slots in one B1itz instance. For a root commitment C_0, call a packet valid if it passes the root check, CoeffCheck, SlotCheck, the corresponding $\mathrm{BatchVerifyEval}$ check, and, for local-share or reconstruction packets, the corresponding LinkCheck.

Lemma 1 (Binding of accepted packets)**.** *Except with negligible probability, every root commitment C_0 accepted by a correct party determines a unique batch of polynomials*

$$\mathcal{F}^{C_0} = \{F_1^{C_0}, \ldots, F_L^{C_0}\}.$$

For $k \in [N]$, $F_k^{C_0}$ has degree t. For $k \in \{N+1, \ldots, L\}$, $F_k^{C_0}$ has degree $2t$ and constant term 0. Moreover, every valid local-share or reconstruction packet for point x contains scalar shares equal to

$$(F_1^{C_0}(x), \ldots, F_L^{C_0}(x)).$$

Proof. The vector commitment binds the root C_0 to one committed vector containing $\widehat{\mathcal{F}}$ and $\hat{S}$, except with negligible probability. Under DLog hardness and independently sampled CRS bases, $\mathrm{BatchCommit}$ is binding, so $\widehat{\mathcal{F}}$ binds each slot to one coefficient vector and $\hat{S}$ binds the receiver-specific degree-t polynomials. The pairing-based CoeffCheck forces the common-base coefficient encodings $\mathsf{Enc}(\mathcal{F})$ to encode the same coefficient vectors as $\widehat{\mathcal{F}}$. Therefore, any coefficient test performed on $\mathsf{Enc}(\mathcal{F})$ is sound for the committed polynomials. Finally, SlotCheck enforces the public slot layout: regular slots have zero coefficients in degrees $t+1, \ldots, 2t$, and high-threshold slots have zero constant coefficient.

Consider an accepted local-share or reconstruction packet for point x with scalar vector $v = (v_1, \ldots, v_L)$. Evaluation binding of $\mathrm{BatchVerifyEval}$ gives that $v_k = S_{k,x}(x)$

for the committed receiver-specific polynomial $S_{k,x}$. At the same time, LinkCheck verifies, for every $k \in [L]$, that

$$g^{v_k} = \prod_{\ell=0}^{2t} \mathsf{Enc}(F_k)[\ell]^{x^\ell}.$$

Since CoeffCheck binds $\mathsf{Enc}(F_k)[\ell]$ to the coefficient encoding $g^{f_{k,\ell}}$ of the committed polynomial F_k, the right-hand side equals $g^{F_k(x)}$. Under DLog hardness, the equality implies $v_k = F_k(x)$. Hence every valid local-share or reconstruction packet from point x contains scalar shares equal to $(F_1(x), \ldots, F_L(x))$. □

Proof of Theorem 1 . We prove the four properties separately.

Privacy. Because $\mathsf{Enc}(\mathcal{F})$ reveals g^{s_k} for each regular constant, privacy is computational with leakage $\mathcal{L}(\mathbf{s}) = (g^{s_1}, \ldots, g^{s_N})$. For uniformly random regular slots this leakage is uniform group elements and hides the scalars under DLog.

Let C be the corrupted points, $|C| \le t$. The simulator samples the CRS as $g_\ell = g^{\rho_\ell}$, so it knows the trapdoor ρ_ℓ. For each slot it samples the corrupted scalar shares and enough extra nonzero points to determine a degree-t polynomial for regular slots, or a degree-$2t$ polynomial with constant 0 for high-threshold slots. It then interpolates coefficient encodings in the exponent: for interpolation set J and encoded values U_x,

$$E_{k,\ell} = \prod_{x \in J} U_x^{\lambda^J_{x,\ell}},$$

using $U_0 = g^{s_k}$ for regular slots and $U_0 = 1_{\mathbb{G}}$ for high-threshold slots. The simulator converts these common-base encodings into the Bulletproof-style commitments by $\widehat{\mathcal{F}}[k] = \prod_{\ell=0}^{2t} E_{k,\ell}^{\rho_\ell}$, so CoeffCheck and SlotCheck hold.

For receiver-specific messages, the simulator samples degree-t polynomials $S_{k,i}$ consistent with the simulated corrupted local shares, commits to the S-batch, and simulates the accepting $\mathrm{BatchVerifyEval}$ proofs for dealer packets and honest-to-corrupt echo messages; corrupt messages are taken verbatim from $\mathcal{A}$. LinkCheck accepts because the local entries are exactly the exponent-interpolated shares above. The simulated view has the same CRS distribution, consistent commitments and encodings, Shamir- distributed corrupted shares, and simulated batched proofs. Hence the adversary learns no scalar information beyond $\mathcal{L}(\mathbf{s})$.

Liveness. If the dealer is correct throughout the sharing stage, all correct parties eventually receive packets with the same root commitment, valid coefficient encodings, and accepting batched evaluation proofs. Hence all correct parties send valid receiver-specific echo messages. Since $n - t \ge 2t + 1$, each correct party eventually receives enough valid echo messages to send ready, and the usual Bracha echo/ready amplification ensures that every correct party eventually receives $2t+1$ ready messages for the same root commitment and completes the sharing stage.

For reconstruction, suppose enough correct parties start reconstructing a slot set $\mathcal{I}$. Let $d_\mathcal{I} = t$ if $\mathcal{I}$ contains only regular slots and $d_\mathcal{I} = 2t$ otherwise. Each such correct party sends a valid reconstruction packet for its point. Since $d_\mathcal{I} < n - t$, every correct party eventually receives $d_\mathcal{I} + 1$ valid packets and interpolates the requested slots.

Correctness. Assume first that the dealer is correct and shares regular values $\mathbf{s} = (s_1, \ldots, s_N)$. The polynomials generated by the dealer have the prescribed constants for the regular slots and constant 0 for the high-threshold slots. By Lemma 1, any correct party that accepts the sharing stage is bound to this same batch of polynomials. Therefore the fixed reconstructed vector is exactly $(s_1, \ldots, s_N, 0, \ldots, 0)$.

Now consider an arbitrary dealer. Once $t + 1$ correct parties complete the sharing stage with the same root commitment C_0, Lemma 1 fixes a unique batch $\mathcal{F}^{C_0}$. Whenever a correct party reconstructs a slot k, every accepted reconstruction packet lies on $F_k^{C_0}$. Since $d + 1$ points uniquely determine a degree-d polynomial, all correct parties reconstruct the same value $F_k^{C_0}(0)$ for that slot.

Completeness. Suppose a correct party completes the sharing stage for root C_0. By the echo/ready argument above, every correct party P_i eventually receives at least $t + 1$ valid receiver-specific echo messages for the same root. For each slot k, such a message from P_j contains a verified point $S_{k,i}^{C_0}(j)$ on the degree-t polynomial indexed by receiver P_i. Hence P_i interpolates $S_{k,i}^{C_0}$ and outputs

$$(S_{1,i}^{C_0}(i), \ldots, S_{L,i}^{C_0}(i)) = (F_1^{C_0}(i), \ldots, F_L^{C_0}(i)),$$

where the interpolation is determined by the accepted batched evaluation proofs in the valid echo messages, and the final equality follows from the LinkCheck performed before output. Thus there exists a fixed batch of committed polynomials $\mathcal{F}^{C_0}$ such that every correct party's output shares are evaluations of this batch at its party index. By Lemma 1, the first N polynomials have degree t and the last B polynomials have degree $2t$ with zero constant term. If the dealer is correct, these polynomials are exactly the dealer's sampled sharing polynomials, so the regular constants equal the dealer's inputs and the high-threshold slots share zero. □

Appendix B. Proofs of ACRDS

Theorem 2 (Termination)**.** *Every honest party P_i eventually outputs $\{[\![r_k]\!]_t\}_{k=1}^{N+B}$ and $\{[\![r_k]\!]_{2t}\}_{k=N+1}^{N+B}$ of* ACRDS(N, B).

Proof. By liveness and completeness of B1itz, if a dealer is correct during sharing, then all correct parties complete the sharing and obtain compatible validated shares. Since there are at least $n - t \ge 2t + 1$ correct dealers, every correct party eventually locally validates at least $2t+1$ B1itz instances and invokes MVBA. By MVBA termination, all correct parties eventually agree on a common subset CS of $2t + 1$ valid dealers. After CS is fixed, the remaining steps in ACRDS are local linear computations and re-indexing, so

every correct party eventually outputs the required shares. □

Theorem 3 (Consistency)**.** *The consistency property of* $\mathsf{ACRDS}(N, B)$ *holds.*

Proof. Let $\widetilde{N} = \lceil N/(t+1) \rceil$, $\widetilde{B} = \lceil B/(t+1) \rceil$, $\widehat{N} = \widetilde{N} + \widetilde{B}$, $\widehat{B} = \widetilde{B}$, and order $CS = \{P_1, \ldots, P_{2t+1}\}$. For dealer P_j, B1itz correctness fixes regular degree-t polynomials $F_{j,1}, \ldots, F_{j,\widehat{N}}$ and high-threshold degree-$2t$ polynomials $Z_{j,1}, \ldots, Z_{j,\widehat{B}}$ with $Z_{j,k}(0) = 0$. For every Vandermonde row ℓ and single-share source k, the derived shares lie on

$$R_{k,\ell}(X) = \sum_{j=1}^{2t+1} M[\ell, j] F_{j,k}(X),$$

a degree-t polynomial; hence any $t+1$ correct shares reconstruct $R_{k,\ell}(0)$. For a double-share source k, define $\widetilde{F}_{j,k} = F_{j,\widetilde{N}+k} + Z_{j,k}$. Then

$$\begin{aligned} \sum_j M[\ell, j] \widetilde{F}_{j,k}(0) &= \sum_j M[\ell, j] (F_{j,\widetilde{N}+k}(0) + Z_{j,k}(0)) \\ &= \sum_j M[\ell, j] F_{j,\widetilde{N}+k}(0), \end{aligned}$$

so the t- and $2t$-side outputs have the same constant. Re-indexing only removes padded coordinates. □

Theorem 4 (Completeness)**.** *If a correct party outputs in an* $\mathsf{ACRDS}(N, B)$ *instance, then there exists a fixed batch of output secrets*

$$(r_1, \ldots, r_{N+B})$$

such that every correct party eventually outputs shares

$$\left\{ [\![r_k]\!]_t^{(i)} \right\}_{k=1}^{N+B} \quad \text{and} \quad \left\{ [\![r_k]\!]_{2t}^{(i)} \right\}_{k=N+1}^{N+B}.$$

Proof. If a correct party outputs, then MVBA has fixed one common dealer set CS. For every dealer in CS, completeness of B1itz guarantees that all correct parties eventually obtain compatible shares for the same committed regular and high-threshold polynomials. The derivation phase applies the same public Vandermonde matrix to this fixed polynomial batch at every correct party. Therefore, the output batch

$$(r_1, \ldots, r_{N+B})$$

is fixed by CS and the committed dealer polynomials, and every correct party eventually outputs the corresponding t-shares and $2t$-shares. □

We separate the proof into two cases. First, if the adversary uses more information from the honest dealers' B1itz transcripts than the public coefficient encodings, then we reduce it to the privacy of B1itz. Second, once all honest B1itz instances are replaced by their simulators, the remaining public information about an honest contribution is only its group encoding, and predicting the scalar breaks DLog.

Theorem 5 (Secrecy)**.** *Assuming DLog hardness and the security of the underlying* B1itz *instances, for any PPT adversary* $\mathcal{A}$ *corrupting a set* $C \subseteq [n]$ *with* $|C| \le t$*, let* view_C *denote the entire view of corrupted parties in an execution of* $\mathsf{ACRDS}(N, B)$*. For every* $j \in [N+B]$ *there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that*

$$\left| \Pr\left[(j, x) \leftarrow \mathcal{A}(\mathsf{view}_C) : x = r_j \right] - \frac{1}{|\mathbb{F}|} \right| \le \mathsf{negl}(\lambda),$$

where r_j *is the* j*-th ACRDS output.*

Proof sketch. By the privacy of B1itz, honest dealers' regular-slot constants are hidden except for their public group encodings; otherwise a standard hybrid over the honest B1itz instances would break B1itz privacy. Every output coordinate contains a nonzero linear contribution from at least one honest dealer in the agreed set CS, so the scalar output remains unpredictable under DLog even though its group encoding is public. For double-shared outputs, adding a high-threshold zero sharing preserves the same plaintext while providing the degree-$2t$ masking. □

Theorem 6 (Complexity)**.** *When the agreement component is instantiated by FIN-MVBA, our ACRDS achieves* $O(n^3)$ *message complexity,* $O(\lambda n^2) + O\left(\frac{n^3}{N+B}\right)$ *amortized communication complexity per output secret, and* $O(n)$ *amortized exponentiation complexity per output secret. It also uses* $O\left(\frac{n^3}{N+B}\right)$ *amortized pairings per output secret.*

We leave the full proof of Theorem 5 and Theorem 6 to our technical report.

Theorem 5. *Assuming DLog hardness and the security of the underlying* B1itz *instances, for any PPT adversary* $\mathcal{A}$ *corrupting a set* $C \subseteq [n]$ *with* $|C| \le t$*, let* view_C *denote the entire view of corrupted parties in an execution of* $\mathsf{ACRDS}(N, B)$*. For every* $j \in [N+B]$ *there exists a negligible function* $\mathsf{negl}(\lambda)$ *such that*

$$\left| \Pr\left[(j, x) \leftarrow \mathcal{A}(\mathsf{view}_C) : x = r_j \right] - \frac{1}{|\mathbb{F}|} \right| \le \mathsf{negl}(\lambda),$$

where r_j *is the* j*-th ACRDS output.*

Proof. Fix a PPT adversary $\mathcal{A}$ corrupting a set $C \subseteq [n]$ with $|C| \le t$.

Replacing honest B1itz *instances.* Let H_0 be the real ACRDS execution. Order the honest dealers arbitrarily, and let H_q be the execution in which the first q honest dealers' B1itz instances are replaced by the simulator guaranteed by Theorem 1, while all remaining parts of ACRDS are executed unchanged. In particular, the outer challenger still runs the same validation logic, MVBA invocation, choice of CS, and linear derivation; the simulator only replaces the transcript of the corresponding B1itz instance and exposes the public coefficient encodings allowed by B1itz privacy.

This composition is valid because the B1itz instances use independent randomness and interact with the rest of ACRDS only through their validated outputs and public validation data. The outer ACRDS code consumes the simulated B1itz outputs through the same interface as in a

real execution, and corrupted dealers' messages are always generated by $\mathcal{A}$.

Suppose that $\mathcal{A}$'s probability of predicting some ACRDS output changes non-negligibly between H_{q-1} and H_q. Then we build a PPT adversary $\mathcal{A}'$ against $\mathsf{B1itz}$ privacy as follows. $\mathcal{A}'$ internally runs $\mathcal{A}$ and simulates the whole ACRDS execution. It embeds its $\mathsf{B1itz}$ challenge as the q-th honest dealer's $\mathsf{B1itz}$ instance, simulates the first $q-1$ honest dealers' $\mathsf{B1itz}$ instances with the $\mathsf{B1itz}$ simulator, and runs or simulates all remaining honest dealers according to the corresponding hybrid. If the embedded challenge is real, $\mathcal{A}$'s view is distributed as in H_{q-1}; if it is simulated, the view is distributed as in H_q. Therefore any non-negligible change in $\mathcal{A}$'s success probability gives $\mathcal{A}'$ a non-negligible advantage against $\mathsf{B1itz}$ privacy: $\mathcal{A}'$ chooses the regular-slot constants used in the embedded $\mathsf{B1itz}$ challenge, so it can compute the corresponding ACRDS output and use $\mathcal{A}$'s correct-guess event as its distinguishing bit. Hence it suffices to analyze the final hybrid, where all honest dealers' $\mathsf{B1itz}$ instances have been replaced by their simulators.

Predicting from public encodings. Consider any output coordinate $\rho \in [N+B]$. This coordinate is obtained, before re-indexing, from some source coordinate k and some row $\ell \in [t+1]$ of the public Vandermonde matrix. Let

$$r_\rho = \sum_{j=1}^{2t+1} M[\ell, j]\, s_{j,k}$$

be the corresponding output secret, where $s_{j,k}$ is the regular-slot constant contributed by dealer $P_j \in CS$. For a double-shared coordinate, the $2t$-side output has the same constant because the added high-threshold polynomial has constant term zero, as shown in the consistency proof.

Let $H = CS \setminus C$ be the honest dealers in the agreed set. Since $|CS| = 2t+1$ and $|C| \le t$, we have $|H| \ge t+1$. The Vandermonde matrix is chosen over distinct nonzero points, so each coefficient $M[\ell, j]$ is nonzero. Hence the honest contribution

$$u_\rho = \sum_{P_j \in H} M[\ell, j]\, s_{j,k}$$

is uniform over $\mathbb{F}$, because it contains a nonzero multiple of at least one independent uniformly sampled honest regular-slot constant. The remaining contribution

$$c_\rho = \sum_{P_j \in CS \cap C} M[\ell, j]\, s_{j,k}$$

is determined by the corrupted dealers' behavior. Thus $r_\rho = u_\rho + c_\rho$.

In the final hybrid, the adversary's view contains the honest scalar constants only through

$$\{g^{s_{j,k}}\}_{P_j \in H}$$

and through Shamir evaluations at corrupted points, which are independent of the constants. Therefore the only public information about u_ρ is its group encoding

$$g^{u_\rho} = \prod_{P_j \in H} \left(g^{s_{j,k}}\right)^{M[\ell,j]}.$$

Equivalently, using all public coefficient encodings, the adversary can compute the group encoding g^{r_ρ}, but not the scalar r_ρ.

If a PPT adversary predicts r_ρ with non-negligible advantage over $1/|\mathbb{F}|$ in this final hybrid, then it computes the discrete logarithm of the public group element g^{r_ρ} with non-negligible advantage. More explicitly, a DLog solver guesses the target coordinate ρ and one honest dealer $P_h \in H$, losing only a polynomial factor. It embeds its challenge $G = g^x$ as the public encoding of the target output coordinate by programming P_h's constant-coefficient encoding in the simulated $\mathsf{B1itz}$ transcript; this is possible because $M[\ell, h] \neq 0$. The programming sets

$$\left(g^{s_{h,k}}\right)^{M[\ell,h]} = G \cdot \left(\prod_{P_j \in CS,\, j \neq h} \left(g^{s_{j,k}}\right)^{M[\ell,j]} \right)^{-1},$$

so that the public encoding of the final output coordinate is exactly $g^{r_\rho} = G$. The programmed transcript is distributed as in the final hybrid, because the $\mathsf{B1itz}$ simulator can generate the scalar shares delivered to corrupted parties together with coefficient encodings that are consistent in the exponent, without knowing the programmed constant coefficient as a scalar. If $\mathcal{A}$ outputs the target scalar, the solver outputs the same value and obtains x. Thus such a predictor breaks DLog. Therefore, for every fixed coordinate ρ,

$$\Pr\left[(\rho, x) \leftarrow \mathcal{A}(\mathsf{view}_C) : x = r_\rho\right] \le \frac{1}{|\mathbb{F}|} + \mathsf{negl}(\lambda).$$

A union bound over the polynomially many output coordinates gives the theorem. □

Theorem 6. *When the agreement component is instantiated by FIN-MVBA, our ACRDS achieves $O(n^3)$ message complexity, $O(\lambda n^2) + O\left(\frac{n^3}{N+B}\right)$ amortized communication complexity per output secret, and $O(n)$ amortized exponentiation complexity per output secret. It also uses $O\left(\frac{n^3}{N+B}\right)$ amortized pairings per output secret.*

Proof. The protocol consists of three phases: commitment, agreement, and derivation. All amortized costs in this proof are per ACRDS output secret; one $\mathsf{ACRDS}(N, B)$ invocation outputs $N+B$ secrets. Let

$$\widehat{N} = \left\lceil \frac{N}{t+1} \right\rceil + \left\lceil \frac{B}{t+1} \right\rceil, \qquad \widehat{B} = \left\lceil \frac{B}{t+1} \right\rceil,$$

so $L = \widehat{N} + \widehat{B}$.

In the commitment phase, every party acts once as a dealer and invokes one $\mathsf{B1itz}$ instance with $\widehat{N}$ regular slots and $\widehat{B}$ high-threshold zero slots, i.e., $L = \widehat{N} + \widehat{B}$ slots in total. Thus ACRDS runs n parallel $\mathsf{B1itz}$ instances. By the complexity of $\mathsf{B1itz}$, one L-slot $\mathsf{B1itz}$ instance has $O(n^2)$ messages, $O(\lambda n^2 L)$ communication, $O(nL)$ exponentiations, and $O(n^2)$ pairings. Since ACRDS invokes one such instance for each dealer, the commitment phase contributes

$$O(n^3)$$

messages and

$$O(\lambda n^3 L)$$

communication in total, and $O(n^2 L)$ exponentiations in total. It also contributes $O(n^3)$ pairings in total.

In the agreement phase, ACRDS invokes one MVBA instance. With FIN-MVBA, this phase has $O(n^3)$ messages and $O(n^3)$ communication. As in the main text, the computation cost of the concrete MVBA/common-coin implementation is accounted for by that component and is not included in the ACRDS validation cost below.

In the derivation phase, parties only apply a public Vandermonde matrix and add the high-threshold zero shares to the corresponding regular shares. This phase sends no messages and uses only field operations.

Combining the three phases gives total message complexity

$$O(n^3) + O(n^3) = O(n^3).$$

Since one invocation outputs $N + B$ random secrets, the amortized communication complexity is

$$\frac{O(\lambda n^3 L) + O(n^3)}{N+B} = O\left(\lambda n^3 \cdot \frac{L}{N+B}\right) + O\left(\frac{n^3}{N+B}\right).$$

Since $t = \Theta(n)$ under the optimal-resilience regime and

$$L = \left\lceil \frac{N}{t+1} \right\rceil + 2\left\lceil \frac{B}{t+1} \right\rceil = O\left(\frac{N+B}{t}\right)$$

for the batched parameter range, we have $L/(N+B) = O(1/n)$. Hence the communication bound simplifies to

$$O(\lambda n^2) + O\left(\frac{n^3}{N+B}\right).$$

For exponentiations, the dominant cost is the n parallel B1itz instances. Using the amortized computation cost of B1itz, the total exponentiation cost is $O(n^2 L)$. Across all dealers and amortized over $N + B$ outputs, this is

$$O\left(n^2 \cdot \frac{L}{N+B}\right) = O(n).$$

For pairings, the only pairing cost in ACRDS comes from the cached CoeffCheck calls inside the n B1itz instances. Since each CoeffCheck is batched across all L slots but still contains $O(n)$ pairings, the n B1itz instances use $O(n^3)$ pairings in total. Amortized over $N + B$ outputs, this is

$$O\left(\frac{n^3}{N+B}\right)$$

pairings per output. This matches the theorem statement in the main text. □

Appendix C. Proofs for TRIO

Fix one TRIO invocation. For each $k \in [D]$, let $A_k(X)$, $B_k(X)$, $R_k(X)$, and $\widetilde{R}_k(X)$ denote the polynomials fixed by the underlying ACRDS output for $[\![a_k]\!]_t$, $[\![b_k]\!]_t$, $[\![r_k]\!]_t$, and $[\![r_k]\!]_{2t}$, respectively. Thus $A_k(0) = a_k$, $B_k(0) = b_k$, and $R_k(0) = \widetilde{R}_k(0) = r_k$. Define

$$P_k(X) = A_k(X)B_k(X) - \widetilde{R}_k(X), \quad d_k = P_k(0) = a_k b_k - r_k.$$

Theorem 7 (Validity)**.** *Except with negligible probability, every correct party that outputs in TRIO holds degree-t shares*

$$\{([\![a_k]\!]_t, [\![b_k]\!]_t, [\![c_k]\!]_t)\}_{k=1}^{D}$$

of values satisfying $c_k = a_k b_k$ for every $k \in [D]$.

Proof. ACRDS consistency fixes the same $A_k, B_k, R_k, \widetilde{R}_k$ and validation data for all correct parties. For an accepted opening vector $\mathbf{p}_j$, define

$$\Delta_k = p_k^{(j)} + \widetilde{R}_k(j) - A_k(j)B_k(j).$$

The random-linear-combination pairing check verifies $\sum_k \gamma_k \Delta_k = 0$. If some $\Delta_k \neq 0$, this nonzero linear equation in random γ_k's holds with probability at most $1/|\mathbb{F}|$; union bounding over senders, every accepted vector satisfies $p_k^{(j)} = P_k(j)$ except negligibly. Since $\deg P_k \leq 2t$, interpolating $2t+1$ accepted vectors reconstructs $d_k = P_k(0) = a_k b_k - r_k$. Each party outputs $[\![c_k]\!]_t^{(i)} = d_k + [\![r_k]\!]_t^{(i)}$; adding a public constant to a degree-t sharing of r_k yields a degree-t sharing of $a_k b_k$. □

Theorem 8 (Secrecy of TRIO)**.** *Assuming the secrecy of ACRDS and CDH hardness in the source group, for any PPT adversary corrupting at most t parties, the adversary's view in a TRIO execution can be simulated from the corrupted parties' output shares, the public validation data, and the public masked openings $\{d_k\}_{k\in[D]}$. In particular, before a triple is consumed by the online MPC protocol, its scalar values a_k and b_k remain computationally hidden and uniformly random from the adversary's view.*

Proof. Fix a PPT adversary $\mathcal{A}$ corrupting a set C with $|C| \leq t$. We show that any non-negligible advantage in predicting an unconsumed scalar a_k or b_k yields either an adversary against ACRDS secrecy or an algorithm that violates the source-group hardness assumption.

Replacing the ACRDS call. Consider two experiments. In H_0, $\mathcal{A}$ interacts with a real TRIO execution. In H_1, the single ACRDS$(2D, D)$ call in Line 02 is replaced by the ACRDS simulator, while the remaining TRIO steps are generated from the resulting shares, public validation data, and corrupted-party messages exactly as in the protocol.

If $\mathcal{A}$'s success probability changes non-negligibly between H_0 and H_1, we construct an adversary $\mathcal{A}_{ACRDS}$ against ACRDS secrecy. The adversary $\mathcal{A}_{ACRDS}$ embeds its challenged ACRDS instance as Line 02 of TRIO, forwards all ACRDS messages between the challenge and $\mathcal{A}$, and runs the non-ACRDS part of TRIO around that instance: corrupted parties' local products are taken from their delivered shares, honest opening messages are generated according to the fixed ACRDS output in the real branch or by the simulator in the simulated branch, and the same vdata is used in the pairing checks. Thus any distinguisher for these

two TRIO experiments distinguishes the real and simulated ACRDS executions, contradicting the secrecy of ACRDS. Hence it remains to analyze H_1.

Information outside ACRDS. In H_1, the scalar information delivered by ACRDS to the corrupted parties consists only of at most t Shamir evaluations of each $A_k, B_k, R_k, \widetilde{R}_k$, together with public coefficient encodings in vdata. The additional public values introduced by TRIO are the masked openings of $P_k = A_k B_k - \widetilde{R}_k$. Since $\widetilde{R}_k$ is a fresh random degree-$2t$ sharing polynomial, P_k is a one-time pad of the product polynomial $A_k B_k$. Equivalently, for any fixed A_k, B_k and any degree-$2t$ polynomial P_k, there is a unique $\widetilde{R}_k = A_k B_k - P_k$, and the distribution of $\widetilde{R}_k$ is uniform. Therefore the public $d_k = P_k(0)$ and the accepted evaluations of P_k are statistically independent of the scalars a_k, b_k, except for the group-encoded relation already present in vdata.

The remaining information about a_k, b_k is therefore contained in source-group encodings such as g^{a_k}, g^{b_k}, and, because d_k is public, $g^{a_k b_k} = g^{r_k + d_k}$. These encodings expose a Diffie–Hellman relation but not the scalar exponents. We state the theorem under CDH hardness; in the scalar-prediction step below we use the implied DLog hardness, since any efficient DLog solver would immediately solve CDH.

Suppose that in H_1, $\mathcal{A}$ predicts some unconsumed a_k or b_k with non-negligible advantage. A DLog solver given a challenge $G = g^x$ guesses the target coordinate k and whether $\mathcal{A}$ will output a_k or b_k. For the case of a_k, it programs $g^{a_k} = G$, samples b_k and d_k as known field elements, and sets $g^{r_k} = G^{b_k} g^{-d_k}$. It also samples the corrupted parties' Shamir evaluations and computes all required coefficient encodings by interpolation in the exponent, exactly as in the ACRDS simulation: the constant term is represented by G, while all programmed corrupted evaluations are known scalars. The degree-$2t$ mask encodings are programmed so that the public relation $g^{d_k} g^{r_k} = g^{a_k b_k}$ and all pairing checks hold. All other coordinates are sampled honestly. If $\mathcal{A}$ outputs the scalar a_k, the solver outputs the same value as x. The case of b_k is symmetric.

Thus a non-negligible predictor yields a non-negligible DLog solver in the source group, contradicting the DLog hardness implied by the CDH assumption. Combining this with the ACRDS-hybrid argument shows that the adversary's advantage in predicting any unconsumed a_k or b_k is negligible. □

Theorem 9 (Complexity of TRIO)**.** *TRIO invokes one* $\mathsf{ACRDS}(2D, D)$ *instance and sends D field elements and the identifier of* vdata *to each party. It has $O(n^3)$ message complexity per invocation, $O(n^3/D)$ amortized message complexity,*

$$O(\lambda n^2) + O\left(\frac{n^3}{D}\right)$$

amortized communication complexity, $O(n)$ amortized exponentiation complexity, and

$$O\left(\frac{n^3}{D} + n^2\right)$$

amortized pairings. The amortized cost is computed as the cost to generate each Beaver triple. The $O(n^2)$ term comes from the batched product-opening verification, where each receiver checks all D product equations from one sender using one random linear combination.

Proof. All amortized costs in this proof are per Beaver triple. The protocol invokes one $\mathsf{ACRDS}(2D, D)$ instance, which outputs $3D$ shared secrets. Substituting $N = 2D$ and $B = D$ in Theorem 6, and absorbing the constant factor between $3D$ outputs and D triples, the ACRDS protocol has $O\left(\frac{n^3}{D}\right)$ messages, $O(\lambda n^2) + O\left(\frac{n^3}{D}\right)$ communication per triple, $O(n)$ amortized exponentiations, and $O\left(\frac{n^3}{D}\right)$ amortized pairings.

After ACRDS completes, each party broadcasts one vector of D field elements, plus an identifier or cached copy of vdata. This adds $O(n^2)$ messages and $O(\lambda D n^2)$ communication in total, i.e.,

$$O\left(\frac{n^2}{D}\right) \quad \text{messages and} \quad O(\lambda n^2) \quad \text{communication}$$

per triple. For each sender, a receiver verifies all D product equations using one random linear combination. The resulting batched equation contains $D + 2$ pairings, so over all sender-receiver pairs the verified opening contributes $O(n^2 D)$ pairings per invocation, or $O(n^2)$ amortized pairings per triple. Forming the batched products costs $O(nD)$ exponentiations per party, or $O(n)$ amortized exponentiations per triple.

The final interpolation uses the same $2t + 1$ reconstruction points for every $k \in [D]$. Each party computes the Lagrange coefficients once and reuses them for all D masked products, so this step only adds local field operations. Combining the ACRDS cost and the verified opening cost gives the claimed message, communication, exponentiation, and pairing bounds. □