DumboMix: Robust Asynchronous Anonymous Broadcast Made Practical

Wei Tang [1] Hanwen Feng [2] Jiliang Li [1] Yuan Lu [3] Qiang Tang [4]

[1] *Xi'an Jiaotong University, {tangwei2023, jiliang.li}@xjtu.edu.cn*
[2] *University of Canterbury, hanwen.feng@canterbury.ac.nz*
[3] *Agency for Science, Technology and Research, lu_yuan@a-star.edu.sg*
[4] *The University of Sydney, qiang.tang@sydney.edu.au*

Abstract

We present a practical framework DumboMix for asynchronous anonymous broadcasts with *guaranteed output delivery* (G.O.D., a.k.a. robustness), enabling a set of n servers to privately solicit N messages from distinct clients, such that these messages remain *secret* until they are revealed in uniform random order. Here, asynchronous G.O.D. ensures that *all* messages are eventually randomly mixed despite (i) arbitrary malicious behaviors by up to $\lfloor (n-1)/3 \rfloor$ Byzantine servers and (ii) unpredictable network delays and jitters.

At the core of DumboMix, we first propose a couple of practical mixing methods DumboMix1 and DumboMix2 for Shamir-secret-shared multi-party computation (MPC) over $\mathbb{Z}_p$, along with their server-optimized variants. When randomly mixing N messages, their online phases require only $O(1)$ multiplicative depth, expected $O(N^2)$ scalar multiplications (between public and shared values), and up to $O(N)$ MPC multiplications (between shared values). Moreover, assuming a robust underlying MPC framework, they guarantee that *all* revealed inputs are uniformly shuffled. In contrast, existing techniques fail to achieve all these performance and functionality features: The DC-net variant Blinder (CCS'20) may reveal a non-negligible fraction of inputs without shuffling them; Butterfly switching networks in secret-shared MPC (CCS'19) incur $O(\log^2 N)$ multiplicative depth; RabbitMix (Security'24) requires $O(N^2)$ MPC multiplications; and PowerMix (CCS'19) incurs $N^3/2$ scalar multiplications.

We finally implement our new mixing methods within DumboMPC++, our computation-optimized implementation of the state-of-the-art robust AMPC framework DumboMPC (Security'25), which provides more concretely efficient offline preprocessing while preserving asynchronous G.O.D. and optimal resilience. We then conduct extensive evaluations with $n = 4$ to 31 servers under varying network settings, revealing that our new mixing methods achieve 44.8–65.9× (resp. 37.1–52.7×), 4.8–7.1× (resp. 3.9–5.5×), and 2.7–4.0× (resp. 5.1–7.2×) speedups over RabbitMix, PowerMix, and the butterfly switching network, respectively, when shuffling 1024 messages in LAN (resp. WAN).

1 Introduction

Internet traffic exposes rich metadata that can substantially compromise end-user privacy even when message contents are encrypted [25, 40, 49]. To mitigate such metadata leakage, a variety of anonymous communication systems, pioneered by Chaum's works of mix-nets [23] and DC-nets [21], have been proposed to prevent adversaries from linking network messages to their senders under global surveillance and traffic analysis. A mix-net service enables the delivery of a list of messages to a few recipients, such that any recipient cannot link any received message to its originating sender [22, 53–55]. On the other hand, a DC-net service, in modern terms [6, 26–28, 33, 37, 38], can be viewed as a special-purpose multi-party computation (MPC) in which a set of servers jointly emulate an *anonymous broadcast* (AB) channel, allowing every client to publish a message to the public so the sender's identity remains hidden among all honest clients.

Necessity of *robust* anonymous broadcast resilient to service disruptions. Recently, efficient constructions of AB via (secret-shared) MPC [4, 20, 24, 33, 52, 57, 58] have received renewed attention. This interest is due not only to their sender-to-public communication model, which makes them well-suited as network-layer anonymity solutions to blockchains [62, 63], but also to their ability to explicitly target adversarial deployment settings by achieving strong robustness guarantees. In particular, they can be instantiated by securely evaluating a mixing circuit within certain (secret-shared) MPC frameworks with guaranteed output delivery (G.O.D.) [4, 29, 58], thereby realizing a robust MPC-as-a-Service (MPCaaS) for message mixing that remains resilient to a fraction of malicious servers. In contrast, many prior anonymous communication systems, including both mix-nets [22, 53, 54] and DC-net-based approaches [6, 26, 27, 37, 38], do not provide *comparable* G.O.D. As a result, even a single malicious participant (server) can drop selected clients' messages and/or force protocol restarts (even if the underlying network is still reliable), thereby potentially blocking meaningful communication for certain clients or even halting the entire system.

Further challenged by the adversarial deployment environment with unpredictable delay and jitter. Moreover, in censorship-prone wide-area networks, a strong asynchronous adversary may arbitrarily delay and reorder protocol transcripts, causing MPC protocols that are robust under synchrony assumptions [11, 29, 30, 42, 44, 47] to lose guaranteed output delivery and, in some cases, even weaken privacy guarantees. In particular, their confidentiality and correctness may rely on synchrony to detect faulty or unresponsive servers and then "eject" them (e.g., removing them from the system), a mechanism that an asynchronous adversary can exploit to eliminate honest parties, thereby compromising privacy or manipulating the output.[1] These observations highlight that asynchronous MPC (AMPC) requires non-trivial redesigns [1, 43] rather than simple adaptation from the synchronous setting, thereby significantly complicating the *practical* implementations of robust AMPC frameworks (with optimal resilience) [8, 58, 67]. In particular, even with preprocessed Beaver triples, online-phase multiplications in an optimal-resilience AMPC framework may still require polynomial interpolation (or even error correction) to tolerate Byzantine/missing shares [58], making multiplication-intensive nontrivial functions, such as random mixing, costly to evaluate.

An overlooked privacy risk in asynchrony caused by weaker ABs that fail to *mix all revealed messages*. Besides the robustness preventing malicious servers from halting an entire AB protocol, there remain two seemingly reasonable design choices: (i) strictly apply a random permutation to shuffle *all* messages solicited from clients [24, 58]; or (ii) relax to randomly mix only a *subset* of messages, while allowing the remaining messages to be revealed without being mixed (e.g., two clients' messages are returned by AB in aggregated form, such as their sum, rather than individual messages). The second relaxed approach can be appealing, since it could be less complex and thus improve efficiency [4] as a natural extension of DC-net–style designs. However, this relaxation introduces a subtle but considerable privacy risk of "re-attempt linkability" in the asynchronous network: messages that are revealed without being shuffled may become linkable across multiple subsequent AB instances. [2]

To exemplify the practical implications of this attack, Appendix C presents a concrete adversary in an asynchronous network: If a message is exposed without being mixed in one AB instance and is resubmitted to subsequent ABs for mixing, the adversary can almost surely identify its sender, just by selectively delaying the clients' messages and thereby controlling which senders are solicited to be mixed in each subsequent AB instance, which under Blinder's system parameters, finally reveals the senders of at least 4% of messages.

Putting together the above desiderata, we focus on practical constructions of the following *anonymous broadcast* protocol that achieves both *robustness* (to mix all solicited messages) and *optimal Byzantine resilience* in an *asynchronous network*.

Definition 1 (Asynchronous Anonymous Broadcast (AAB), informal). *We consider* **robust AAB** *as a special-purpose MPC protocol in the client–server model, where a set of* $(N+F)$ *clients each privately submit a message, such that at least* N *such messages are privately committed to a set of* n *servers. The servers then jointly reveal, and* **only** *reveal,* **all** *committed messages as output in* **a uniformly random order** *(i.e., fully shuffled), while ensuring guaranteed output delivery despite a fully asynchronous network and Byzantine behaviors of up to* $\lfloor (n-1)/3 \rfloor$ *corrupt servers and up to* F *corrupt clients.*[3]

1.1 State-of-the-art and Remaining Issues

The standard approach to constructing a robust AAB (Def. 1) is first to build a robust AMPC framework [58, 67] for general computation, and then evaluate a mixing circuit within it [24, 58]. However, all existing constructions (and plausible candidates) suffer from significant performance bottlenecks. Their inefficiencies stem from their high asymptotic (online-phase) complexities. Specifically, they suffer from at least one of the following limitations in the critical online phase:

- *Ever-growing Round Complexity*: Certain mixing circuits, such as a butterfly-style switch net [58], have a poly-logarithmic circuit depth (e.g., $O(\log^2 N)$ for N input messages). This necessitates a logarithmic number of communication rounds. Consequently, the online latency of AAB services will strictly increase as the system scales to achieve meaningful anonymity, regardless of the bandwidth or computational resources invested.
- *Cubic Scalar Multiplication Complexity*: Scalar multiplication is a standard MPC operation wherein a secret-shared value is multiplied by a public scalar. Some mixing methods, such as PowerMix [58], require each party to perform $O(N^3)$ such operations in the online phase. This cubic overhead is prohibitively expensive, even if considering that they are executed locally.
- *Quadratic MPC Multiplication Complexity*: MPC multiplication, which involves multiplying two secret-shared values, is typically the most expensive operation in robust MPC. Nevertheless, protocols like RabbitMix [24]

[1] A similar concern arises for many *robust* anonymous communication systems, such as Trellis [55], which achieve robustness through a blame-elimination-recovery mechanism that fundamentally relies on synchrony (cf. Section III of [55]), such that an asynchronous network may cause an adversarially delayed honest server to be treated as faulty and be ejected.

[2] Note that in practice, application-level messages input to AB protocols are themselves required to be anonymous (i.e., unlinkable) at the application layer for each client [62, 63] (e.g., confidential transactions [15, 48, 64]). Nevertheless, re-attempt linkability still arises if an anonymous transaction fails to be delivered and is subsequently resubmitted, unless the client additionally re-randomizes the confidential transaction before each retransmission.

[3] Note folklore impossibilities: (i) robust AMPC cannot be realized with resilience $\geq n/3$; (ii) AMPC cannot guarantee the collection of all participants' inputs, and given F malicious clients among $N+F$, soliciting N inputs for computation is best possible.

require $O(N^2)$ such online-phase operations, constituting a severe performance bottleneck. The serious impact on latency is shown in Fig. 1.

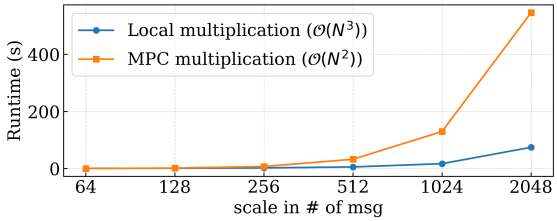


Figure 1: Large latency caused by $O(N^3)$ local scalar mul. (PowerMix) or $O(N^2)$ MPC mul. (RabbitMix): executed within an AMPC framework hbMPC [58] (4 servers, LAN).

Although efficient mixing methods such as Blinder [4] exist, they do not provide guaranteed output delivery and remain vulnerable to "re-attempt linkability" as briefly mentioned. Consequently, they cannot be directly used to realize an AAB service without additional application-level mechanisms that carefully rerandomize each message upon resubmission.

In light of the above limitations lying in the state of the art,[4] enabling a practical and robust AAB service still calls for more efficient mixing methods that not only incur asymptotically lower evaluation costs within robust AMPC frameworks, but also ensure robust delivery of all solicited messages such that no solicited message is exposed unless it is mixed.

Table 1: AABs' online cost for N messages (i.e., applying each mixing circuit in robust AMPC with optimal resilience).

	Scalar mul. *	**MPC mul.** *	**Mul. depth**	***Robust* delivery**	**Client's msg size**
SwitchNet [58]	$\tilde{O}(N)$	$\frac{N\log^2 N}{2}$	$\log^2 N$	✓	$O(1)$
PowerMix† [58]	$O(N^3)$	-	$O(1)$	✓	$O(1)$
RabbitMix† [24]	$O(N^{1.5})$	N^2	$O(1)$	✓	$O(\sqrt{N})$
Blinder-1‡ [4]	$O(N^2)$	$3N$	$O(1)$	✗	$O(N)$
Blinder-2‡ [4]	$O(N^{1.5})$	N^2	$O(1)$	✗	$O(\sqrt{N})$
DumboMix1 (§4.1)	$O(N^2)$	$2N$	$O(1)$	✓	$O(N)$
DumboMix2 (§4.2)	$O(N^2)$	-	$O(1)$	✓	$O(1)$

† RabbitMix can be viewed as a special variant of PowerMix where each client secretly shares $\sqrt{N}$ selected powers of its message rather than just sharing the message itself and computing all powers during the online phase as PowerMix.
‡ Blinder presents two variants where each client secretly shares $O(N)$ and $O(\sqrt{N})$ elements from $\mathbb{Z}_p$, respectively; in expectation, $0.05*N$ messages are revealed without being shuffled in both variants.
* "Scalar mul." is the number of local multiplications performed by each server, while "MPC mul." refers to multiplication gates within the mixing circuit.

1.2 Our Contribution

Bearing the limitations of existing robust AAB constructions and candidates, we propose DumboMix, a practical AAB framework with G.O.D. and optimal resilience. In greater detail, our contributions can be summarized as follows:

- **Formal modeling of AAB emphasizing critical robustness aspects**. To capture all security aspects required for mission-critical anonymous broadcasts, we introduce, for the first time, an ideal functionality for robust AAB that formalizes resilience to DoS disruptions and native prevention of re-attempt linkability. In particular, our formalism (cf. §3) captures the following robustness properties even in the presence of adversarial network delays and Byzantine participants:
  - *Best-possible delivery rate*. To counter service disruptions, for any $N+F$ clients containing F corrupt clients, the protocol must process message submissions from at least N distinct clients for mixing;
  - *Reattempt-ability*. Moreover, to address the "re-attempt linkability" threat (as demonstrated in Appendix C), messages must remain private unless they are successfully mixed.
- **More efficient mixing methods for secret-sharing-based MPC over $\mathbb{Z}_p$**. As our core contribution, we design two fast mixing circuits for MPC-based AAB protocols, DumboMix1 and DumboMix2 (cf., §4), that efficiently shuffle messages from thousands of clients—a scale critical for achieving meaningful anonymity. Table 1 compares their efficiency with existing mixing methods. DumboMix1 achieves $O(1)$ multiplicative depth, $O(N)$ MPC multiplications between shared values, and an expected $O(N^2)$ scalar multiplications between public and shared values. DumboMix2 further optimizes DumboMix1 via additional preprocessing, reducing each client's field elements to share from $O(N)$ to $O(1)$, while also eliminating the need for online MPC multiplications between shared values.
- **Performance-optimized variants of our mixing methods**. We also present DumboMix1* and DumboMix2*, which improve the server-side concrete efficiency of our two mixing circuits, DumboMix1 and DumboMix2, respectively, yielding an approximately 40% reduction in their online-phase latency, as demonstrated by our experimental evaluations in §5. Our highly-optimized DumboMix1* and DumboMix2* (as well as their base versions) can be viewed as a new family of arithmetic mixing circuits over $\mathbb{Z}_p$ and may be of independent interest for secret-sharing-based MPC mixing under other communication models beyond asynchronous settings.
- **Performance-optimized implementation and extensive evaluations**. Finally, we implement our AAB mixing methods within a robust AMPC framework with optimal resilience, building on DumboMPC [67] and optimizing its offline preprocessing for better concrete efficiency. Atop this framework (called DumboMPC++ by us), we implement and deploy our performance-optimized mixing circuits DumboMix1*

[4] A more detailed discussion of existing AAB, candidate approaches, and their limitations is provided in Appendix A and §2.1.

and DumboMix2*, and experimentally compare them with state-of-the-art mixing methods under identical AMPC settings, using up to 31 cloud servers in both LAN and WAN environments. When mixing 1024 messages, they achieve at least 44.8× (resp. 37.1×), 4.8× (resp. 3.9×), and 2.7× (resp. 5.1×) speedups over RabbitMix [24], PowerMix [58], and the butterfly switching network [58], respectively, in LAN (resp. WAN).

2 Challenges and Our Techniques

Following most MPCaaS [51, 58], our AAB operates in the online/offline paradigm: In the offline phase, various forms of correlated randomness *independent* of the clients' inputs are preprocessed; in the online phase, a mixing circuit is evaluated to reveal the clients' inputs in a random order, consuming the preprocessed correlated randomness. This section reviews existing approaches, highlights their remaining challenges, and summarizes our techniques for improving the efficiency of both the online mixing circuit and the offline preprocessing.

2.1 Prior Art and Remaining Challenges

We begin by reviewing a line of existing works to outline the core technical challenges underlying our design objectives.

DC-Net: Non-robust anonymous broadcasting. The seminal work on DC-net [21] provides a simple yet elegant method for anonymously broadcasting a message. In the client-server MPC paradigm, a DC-net can be viewed as: in each session, only one client, denoted by i, among the total N clients, is permitted to send a message m_i. This client secretly shares m_i with the servers, while the non-sender clients (i.e., for $j \in [N], j \neq i$) share $m_j = 0$. Finally, the servers jointly reveal $m = \sum_{j\in[N]} m_j = m_i$ to finish the broadcast. Sender is anonymous as m reveals no information about identity i.

Despite its simplicity and efficiency, the original notion of DC-net has an obvious drawback: only one client is allowed to send a message per session, and if two or more clients submit messages simultaneously, either accidentally or due to malicious behavior, the protocol fails to deliver any message, meaning it lacks robustness. Historically, DC-nets were extended to support mixing of multiple clients' messages via a scheduling mechanism: time is divided into a sequence of N DC-net sessions, each session is assigned to (at most) one client to avoid collisions, and the outputs of all sessions are aggregated to form a shuffled message set [27, 28]. However, such an extension inherently relies on network synchrony, and it is unclear how to generalize it to an asynchronous setting. The alternative approach of mix-nets [23] unfortunately necessitates non-constant round complexity.

Algebraic mix: mixing multiple messages at a higher cost. The naive DC-net approach fails in the presence of multiple senders because individual messages $\{m_j\}_{j\in[N]}$ cannot be recovered from the single sum $\sum_{j\in[N]} m_j$. Recent works, such as PowerMix [58] and RabbitMix [24], tackle this problem by computing the summations of the powers of the messages rather than the messages themselves. Specifically, once these summations (denoted by $\sigma_1, \ldots, \sigma_N$) are calculated, the individual messages $\{m_1, \ldots, m_N\}$ can be recovered as roots of the following system of equations:

$$\begin{cases} m_1 + m_2 + \cdots + m_N = \sigma_1 \\ m_1^2 + m_2^2 + \cdots + m_N^2 = \sigma_2 \\ \cdots \quad \cdots \quad \cdots \quad \cdots \quad \cdots \\ m_1^N + m_2^N + \cdots + m_N^N = \sigma_N \end{cases} \tag{1}$$

Here, the sender anonymity is preserved because the set $\{\sigma_1, \ldots, \sigma_N\}$ does not reveal any information regarding which party contributed which message m_i.

Unfortunately, this approach incurs significant overhead, as securely obtaining the summations $\{\sigma_1, \ldots, \sigma_N\}$ first requires calculating $m_j, m_j^2, \ldots, m_j^N$ for each $j \in [N]$ in a *secret-shared* form, which is highly expensive. PowerMix and RabbitMix adopt different tradeoffs for this, resulting in either $O(N^3)$ scalar multiplications (PowerMix) or $O(N^2)$ MPC multiplications (RabbitMix), both of which pose a severe bottleneck in robust AMPC frameworks [58], as Fig. 1 indicates.

Blinder: improved efficiency but without guaranteed delivery. To shuffle multiple messages simultaneously without introducing excessive costs, Blinder [4] takes a different approach to batch multiple DC-net sessions: Each client *randomly* selects one session to send its message and provides shares of zeros in the remaining sessions. Specifically, each client i creates an L-dimensional zero vector M_i, randomly picks an index $j \leftarrow [L]$, assigns $M_j^{(i)} = m_i$ for its input message m_i, and then secretly shares the vector $M^{(i)}$ among all servers.

Ideally, if different clients place their messages in distinct positions within the vector, the servers could simply reveal the summation of all shared vectors, i.e., $\mathbb{T} = \sum_{i\in[N]} M^{(i)}$, where each non-zero entry corresponds to a distinct input. However, ensuring a negligible collision probability requires an excessively large vector (e.g., $L = O(\lambda N^2)$), resulting in prohibitive overhead. Conversely, a smaller L significantly increases the probability of multiple clients selecting the same entry.

Blinder finds a "sweet spot" by configuring $L = 2.75 * N$, which bounds the probability of three or more clients colliding at a single position to less than 5%. It resolves two-client collisions using a strategy similar to PowerMix: in addition to calculating $\mathbb{T} = \sum_{i\in[N]} M^{(i)}$, it also calculates: $\mathbb{T}' = \left(\sum_{i\in[N]} (M_1^{(i)})^2, \ldots, \sum_{i\in[N]} (M_L^{(i)})^2\right)$. The final outputs are the non-zero roots to this system of *quadratic* equations. This approach is more efficient than PowerMix and RabbitMix because it avoids computing higher-order powers from the 3-rd to the N-th. However, its drawbacks are also clear:

- *No guaranteed delivery:* Even if all participants are honest, about 5% messages will fail to be delivered as colliding with three or more other messages, and resubmission of them may enable an adversary to link the sender's identity (cf. an analysis in §C).

- *$O(N)$ client overhead:* Under this scheme, each client must secretly share an $O(N)$-sized vector among the servers. This overhead can be excessive for resource-constrained clients, such as mobile or IoT devices.

Though Blinder provides some mitigation to reduce clients' overhead, which, as we will discuss shortly, lacks efficient instantiations in our setting.

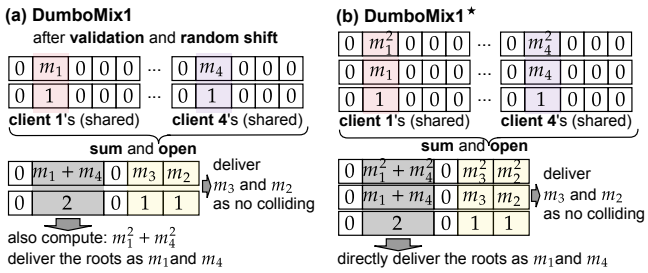


Figure 2: Toy examples a) DumboMix1 and b) DumboMix1⋆

2.2 Our Techniques

As outlined above, two core gaps exist in the literature: (i) ***eliminating the excessive computing cost*** incurred in PowerMix (i.e., *cubic* local mul.) and RabbitMix (i.e., *quadratic* MPC mul.) when calculating up to the N-th power of shared messages, while still guaranteeing the delivery of all messages; and (ii) ***avoiding the $O(N)$ client-side overhead***. Specifically, we aim to address these two gaps in the asynchronous setting, where the adversary can corrupt up to $\lfloor (n-1)/3 \rfloor$ servers and F clients, while maintaining the $O(1)$ round complexity of existing approaches. Note that Blinder cannot guarantee delivery even when all clients are honest.

Hereafter, we present our key technical ideas for addressing these gaps step-by-step. Afterward, we discuss how we leverage asynchronous fault-tolerant protocols, such as consensus and broadcast, to ensure these ideas function properly in an asynchronous Byzantine environment.

Failed initial attempt: Adding "parity bits" in Blinder to restore robust delivery. Recall that Blinder lacks robust delivery because it simply outputs the non-zero roots to the quadratic equation system, so when three or more messages collide, the roots no longer correspond to the original messages. One may suggest a straightforward idea to restore robust delivery in Blinder: each client just encodes its message to share in a specific format, such as $m \,\|\, \mathsf{Hash}(m)$, so that a failure can be detected whenever a root does not satisfy this format. For example, if for some position $j \in [L]$, the non-zero roots for the equation system

$$\begin{cases} M_j^{(1)} + \cdots + M_j^{(N)} = \sigma_j \\ (M_j^{(1)})^2 + \cdots + (M_j^{(N)})^2 = \sigma_j^2 \end{cases} \tag{2}$$

do not satisfy the expected format (i.e., not passing "parity" check), the servers must then calculate σ_j^3, which equals

$$(M_j^{(1)})^3 + \cdots + (M_j^{(N)})^3 = \sigma_j^3, \tag{3}$$

and find the non-zero solutions for the combined equation system of Eq. 2 and Eq. 3. This process would repeat, calculating up to the summation of the N-th power, until appropriately formatted solutions are found. However, this approach incurs significant overhead: (i) the error-detection "parity" bits (e.g., $\mathsf{Hash}(m)$), which are typically several dozen bytes, constitute a substantial overhead relative to the message size, especially in secret-sharing-based AMPC (e.g., HoneyBadgerMPC and DumboMPC), the message space typically supports only 32 bytes; and (ii) the repeated decoding attempts introduce additional rounds (which is costly in WAN).

Step 1 DumboMix1*: robust delivery "almost for-free" by explicitly tracking collisions.* As shown in Fig.2 (a), instead of detecting delivery failures by "parity bits" after the fact, our design develops new techniques to track the number of collisions actively. Specifically, while each client i already shares a L-size vector $M^{(i)}$ containing at most one non-zero item at the j-th entry, denoted by $M_j^{(i)}$, we let the client i additionally share an L-size *one-hot* indicator vector $I^{(i)}$, marking the same position j.

Now, the servers can then jointly reveal the summation of all clients' indicator vectors: $\mathbb{I} = \sum_{i \in [N]} I^{(i)}$. Each entry $\mathbb{I}[j]$ in $\mathbb{I}$ explicitly reveals the exact number of colliding messages at position j. Equipped with this information, the servers can jointly compute the exact required summations via AMPC:

$$\begin{cases} M_j^{(1)} + \cdots + M_j^{(N)} = \sigma_j \\ \ddots \\ (M_j^{(1)})^{\mathbb{I}[j]} + \cdots + (M_j^{(N)})^{\mathbb{I}[j]} = \sigma_j^{\mathbb{I}[j]} \end{cases}$$

The (non-zero) roots to this specific equation system are then delivered as these messages colliding at the position j.

Note that, to prevent malicious clients from sharing vectors in wrong format, the servers need to conduct a couple of checks for each client i: (i) the indicator vector $I^{(i)}$ is genuinely one-hot (containing all 0s except for exactly one entry with "1"), and (ii) the message vector $M^{(i)}$ also contains at most one non-zero entry located at the same position as the "1" in $I^{(i)}$. To enforce these properties, we introduce an efficient input validation phase based on standard Schwartz-Zippel-style polynomial identity tests [4, 65, 75].

In addition, we expect to ensure that all clients (including corrupted ones) place their non-zero entries in their vectors at

random positions, which is crucial for bounding the probability of excessive message collisions at the same position and, consequently, for limiting the computational cost of evaluating the power-summations required to resolve colliding messages. Nevertheless, malicious clients may be coordinated by the adversary to place their messages at the same position, inducing high-order collisions that force the servers to compute up to the $O(N)$-th power of shared messages, thereby incurring asymptotic costs comparable to PowerMix or RabbitMix.

To ensure random placement of non-zero entries in each client's vectors, we employ a simple public re-randomization technique. After clients have already shared their input vectors, servers jointly toss an unpredictable public random $\mathsf{coin} \leftarrow \{0,1\}^*$ [18]. A random *offset* is independently derived for each client i as $H(\mathsf{coin}||i)$, where $H : \{0,1\}^* \to [L]$ is a hash function modeled as random oracle, so all servers can cyclically shift their respective shares of the vectors associated with client i according to the derived *offset*. Because different clients' vectors are independently and randomly shifted, their nonzero entries are placed uniformly at random across L positions. Consequently, the number of collisions at each position closely follows a Poisson distribution, thereby neutralizing any coordinated malicious placement strategy.

Combining all treatments described above, we can obtain DumboMix1 that restores robust delivery in Blinder. Together with a probability analysis to bound the expected orders and the expected numbers of message collisions, we can show that DumboMix1 achieves $O(N^2)$ scalar multiplications and $O(N)$ MPC multiplications (matching those in Blinder), thereby avoiding both $O(N^3)$ scalar multiplications in PowerMix and $O(N^2)$ MPC multiplications in RabbitMix.

Step 2 DumboMix2*: reducing client-side overhead to $O(1)$ from $O(N)$ by additional preprocessing.* Recall that Blinder proposed two approaches to reduce client overhead, both of which are unfortunately expensive in our setting. The first employs distributed point functions [13, 41] to achieve $O(1)$ client overhead, but efficient instantiations capable of tolerating $\lfloor (n-1)/3 \rfloor$ corruptions among n servers are currently unknown. The second requires the servers to perform $O(N^2)$ MPC multiplications; Blinder can afford this only because it assumes a weaker threat model (only $n/4$ server corruptions), where weaker resilience allows efficient MPC multiplications.

We resolve this bottleneck by leveraging correlated randomness generated during preprocessing. Specifically, we present a method allowing each client i to send an $O(1)$-sized public message, from which the servers can *locally* derive two shared L-size vectors $M^{(i)}$ and $I^{(i)}$ (equivalent to those in DumboMix1), satisfying all aforementioned requirements.

During the preprocessing, the servers securely generate N shared one-hot *random* vectors with a length of L, denoted $\{I^{(j)}\}_{j\in[N]}$, alongside N shared one-Hamming-weight *random* vectors also having L items, denoted $\{U^{(j)}\}_{j\in[N]}$. These are generated such that each corresponding pair $I^{(j)}$ and $U^{(j)}$ shares the same non-zero position; moreover, such additional offline preprocessing can be efficient, consuming only $O(\log(L))$ rounds and $O(L)$ MPC multiplications (where $L = O(N)$). During the online phase, when a client i wishes to send an input message m_i, it requests the shares of $U^{(i)}$ from the servers and reconstructs the non-zero value, denoted u_i. The client i then simply broadcasts $u_i^{-1} \cdot m_i$ to all servers. Each server holding a share of the vector $U^{(i)}$ multiplies each entry of its share by $u_i^{-1} \cdot m_i$. This locally yields a share of a new one-Hamming-weight vector, $M^{(i)}$, that embeds m_i at the exact same position as the non-zero entry of $U^{(i)}$. The servers then proceed using the shared vectors $I^{(i)}$ and $M^{(i)}$ exactly as if the client i directly provides them.

Applying the above technique to DumboMix1 results in DumboMix2, reducing client overhead to $O(1)$ by shifting the heavy computation to server-side preprocessing.

Step 3 DumboMix1* *and* DumboMix2**: Improving server's concrete efficiency with the help of clients.* We further optimize concrete performance by carefully balancing the burdens between the clients and the servers. In the solutions above, the servers must jointly compute power sums via AMPC from the provided vectors, which dominates the server-side cost.

With a proper choice of the vector dimension L (still linear in N), the probability of three or more clients colliding at the same position is small. This implies that the servers primarily only need to calculate the summation of the squares of the shared entries, as observed in [4]. As Fig.2 (b) shows, this burden can be alleviated by adopting the idea from [4]. Specifically, each client i shares an additional vector, denoted $\hat{M}^{(i)}$, containing the value m_i^2 at the only non-zero entry. Because clients can calculate m_i^2 in plaintext instantaneously, this optimization incurs only additional client-side communication overhead. Particularly in DumboMix2, this client overhead is marginal, yet it significantly reduces the dominant server-side cost associated with squaring secret-shared values.

This optimization for concrete performance can be applied to both DumboMix1 and DumboMix2 (with appropriate adjustments to input validation). We denote the resulting highly optimized schemes as DumboMix1* and DumboMix2*, respectively. See §4.3 and Appendix H for details.

Integrate fault-tolerant primitives into full-fledged solutions. To provide a complete picture, we now brief how the aforementioned operations, such as the client's sharing/broadcasting and the scalar/MPC multiplications, are securely instantiated in an asynchronous network. The MPC multiplications and scalar multiplications are standard and readily available in most AMPC frameworks [58]. When a client needs to share a value m_i among the servers, it first asks the servers to help it reconstruct a preprocessed randomness r, and then consistently broadcasts $m_i + r$ to all servers using a reliable broadcast (RBC) [14], so every server can locally derive a share of m_i by subtracting its share of r from the received $m_i + r$. Finally, an asynchronous Byzantine consensus proto-

col, specifically Multivalued Validated Byzantine Agreement (MVBA) [17, 34, 59], is executed by the servers to agree on the exact set of clients whose inputs will be mixed.

Soon in §4, we will formally describe the protocols to clarify how these fault-tolerant primitives are orchestrated. Nevertheless, our core technical ideas, DumboMix1, DumboMix2, and their server-side optimization, are also applicable to other secret-sharing-based MPC frameworks over $\mathbb{Z}_p$ (e.g., SPDZ [51]), and are thus not limited to the fully asynchronous instantiations considered in this paper.

3 Problem Formulation and Preliminaries

Notations. Let $[\![x]\!]$ denote a secret sharing of x. We consider n servers, among which $t < n/3$ may be corrupted, and use N and F to denote the numbers of honest and corrupted clients, respectively. Unless otherwise specified, parenthesized superscripts denote client indices, while subscripts denote positions in vectors or sequences. For example, $I_i^{(j)}$ denotes the i-th entry of client j's vector $I^{(j)}$. When no ambiguity, we also write m_i as shorthand for message $m^{(i)}$ of client i.

System participants. We consider *asynchronous MPC* in the client-server model, i.e., the MPC-as-a-service (MPCaaS) setting. The servers jointly provide a robust asynchronous anonymous broadcast (AAB) service (i.e., secure MPC-based shuffling) to the clients. The set of n servers is fixed and denoted by $\mathcal{P} = \{P_i\}_{i\in[n]}$. We assume a standard Public Key Infrastructure (PKI) setup for these servers, meaning their identities and public keys are globally known throughout the system. The clients' identities are known to all servers but not necessarily known to peer clients.

Network model. We assume that every pair of servers is connected by a standard *asynchronous secure channel*, so messages exchanged between any two honest servers remain confidential from third parties, cannot be altered, and are guaranteed to be delivered eventually, although an adversary may arbitrarily delay their delivery. Furthermore, every client is connected to all servers via identical asynchronous secure channels. In practice, such channels can be instantiated using standard real-world network protocols such as TLS.

Adversarial model. We consider a computationally bounded adversary capable of corrupting both servers and clients. In particular, the adversary is allowed to *statically* corrupt up to $t = \lfloor (n-1)/3 \rfloor$ of the n servers. These corrupted servers are *Byzantine*, meaning they may deviate from the protocol specifications and behave arbitrarily under the adversary's control. We remark that the threshold $t = \lfloor (n-1)/3 \rfloor$ is indeed *optimal* for AMPC with guaranteed output delivery. The adversary may also corrupt some clients, who then similarly exhibit Byzantine behavior (e.g., remain silent or send malformed messages as instructed by the adversary). We denote the number of corrupted clients and the number of honest clients by F and N, respectively.

Epoch-based fix-batch service. Due to asynchrony, servers cannot wait for all clients before starting the shuffling service. Instead, they execute a sequence of AAB instances to form an epoch-based service: In each epoch, they start an AAB to process a fixed batch of N messages from distinct clients, and enter the next epoch to start another AAB and process another N-size batch, once the current AAB completes. To realize meaningful sender anonymity, there must be enough inputs among the N inputs originating from honest clients. With F corrupt clients in total, the effective anonymity set (per AAB) has a size of $N - F$. So if F is too large, anonymity degrades accordingly. Addressing this issue requires orthogonal Sybil-resistance techniques, such as proof-of-work [36] or external identity verification and fees.

Design objectives. We aim to design an efficient AAB protocol (i.e., a specialized AMPC protocol for message shuffling) that also achieves optimal resilience and G.O.D. In particular, we put forth the ideal functionality of $\mathcal{F}_{\mathsf{AAB}}$ to formally extend Definition 1 and capture the robustness properties of anonymous broadcast highlighted in the Introduction:

- *Guaranteed delivery ratio*. Every AAB protocol not only terminates, but also delivers at least $N - F$ messages from honest clients.
- *Unlinkability*. In an epoch with N honest clients and F corrupt clients, any PPT adversary can link a message sent by an honest client to its sender with probability at most $\frac{1}{N-F}$ (as all delivered messages sent by honest clients are from "$\perp$" in the adversary's view).
- *Reattempt-ability*. This ensures the privacy of undelivered messages: For any honest client i, if its input message m_i is not delivered by the epoch's AAB protocol, the adversary cannot learn any information about m_i.

Preliminary. Below, we introduce the high-level abstractions of several established cryptographic and distributed primitives in our constructions, deferring details to Appendix B.

Multi-valued Validated Asynchronous Byzantine Agreement (MVBA). In MVBA [3, 17, 34, 59], each honest party P_i provides a *valid* value v_i as the input, and all honest parties eventually output the same valid value v. The validity of values is defined by an external predicate Q, i.e., v is valid, iff $Q(v) = 1$.

Reliable Broadcast (RBC). In a RBC protocol [14, 19], a designated sender broadcasts a value v to all other parties in the network. When the sender is honest, all honest parties eventually receive v. If some honest party outputs a value v', then all honest parties must eventually output the same v'.

Threshold Common Coin. In a threshold common coin protocol [18], all honest parties eventually output a value $r \in \{0,1\}^\kappa$, if all of them invoke $\mathsf{Coin.Get}(\mathsf{ctx})$ for a given public context ctx. Here r cannot be predicted by the adversary better than guessing over $\{0,1\}^\kappa$, before the first honest party invokes $\mathsf{Coin.Get}(\mathsf{ctx})$.

Functionality $\mathcal{F}_{\mathsf{AAB}}$

$\mathcal{F}_{\mathsf{AAB}}$ supports a message space $\mathbb{M}$ that is $\mathbb{Z}_p^*$ or $\mathbb{Z}_p$. (Looking ahead, $\mathbb{Z}_p$ and $\mathbb{Z}_p^*$ will be supported by DumboMix1 and DumboMix2, respectively.) Let $\mathcal{H}$ and $\mathcal{C}$ denote the sets of honest and corrupted clients, respectively, where $|\mathcal{H}| = N$ and $|\mathcal{C}| = F$. $\mathcal{F}_{\mathsf{AAB}}$ interacts with these clients, n servers, and the adversary $\mathcal{S}$; it initializes an empty set $\mathcal{L}^*$, and proceeds as.

- **Input Submission**. Upon receiving a message (i, m_i) from client i for the first time:
  - $\mathcal{L}^* \leftarrow \mathcal{L}^* \cup \{i\}$ and send i to the adversary
  - internally record (i, m_i), where $m_i \in \mathbb{M} \cup \{\perp\}$
- **Input Selection**. After $|\mathcal{L}^*| \geq N$, wait for the adversary to choose a set $\mathcal{L} \subseteq \mathcal{L}^*$ s.t. $|\mathcal{L}| = N$:
  - initialized an empty list $\mathcal{X}$
  - for every $i \in \mathcal{L}$ s.t. $m_i \neq \perp$ (i.e., skip i if $m_i = \perp$):
    - if $i \in \mathcal{H}$: append $(\perp, m_i)$ to $\mathcal{X}$
    - if $i \in \mathcal{C}$: append (i, m_i) to $\mathcal{X}$
- **Mixing**. Wait for $\mathcal{S}$'s approval to execute:
  - sample a uniformly random permutation π over $\mathcal{X}$
  - send $\pi(\mathcal{X})$ to $\mathcal{S}$
  - let msgs be the ordered sequence obtained from $\pi(\mathcal{X})$ by retaining only messages and removing all senders
- **Delivery**. Wait for $\mathcal{S}$ to instruct to delivery:
  - output msgs to every server (at time specified by $\mathcal{S}$)

4 DumboMix: Our AAB Constructions

We present our AAB constructions through four variants. The basic version DumboMix1 uses the client-side random placement and requires $O(N)$ client communication, while DumboMix2 moves placement generation to server-side preprocessing and reduces client communication to $O(1)$. We further introduce two implementation-oriented variants, DumboMix1* and DumboMix2*, improving concrete efficiency by handling two-message collisions directly.

4.1 DumboMix1

We now present the base version DumboMix1, which securely realizes the ideal functionality $\mathcal{F}_{\mathsf{AAB}}$ with a message space $\mathbb{Z}_p$. The protocol costs constant rounds, linear MPC multiplications, and quadratic per-server scalar multiplications.

PREPROCESSING MATERIALS. We instantiate DumboMix1 in the standard asynchronous MPCaaS model described in Section 3, which follows an online–offline paradigm, where an asynchronous offline phase (cf. [67] or Appendix D) continuously generates input-independent resources to support more efficient online computation. Specifically, the following preprocessing materials are required by DumboMix1.

- *Shared randomness.* A sequence of shared random numbers $\{[\![r_1]\!], [\![r_2]\!], \dots\}$, which are used for masking the clients' inputs or for reconstructing common coins.
- *Shared multiplication triples.* A sequence of shared multiplication triples $\{([\![a_1]\!], [\![b_1]\!], [\![c_1]\!]), \dots\}$, which are consumed implicitly in multiplications over shared values.
- *Shared powers.* For each $k \in [2, N]$, a sequence of shared powers of random values $\{([\![\eta_1]\!], [\![\eta_1^2]\!], \dots, [\![\eta_1^k]\!]), \dots\}$, which can be generated from shared randomness through (iterative) MPC multiplications. [5]

INGREDIENTS. In addition to the aforementioned offline preprocessing, DumboMix1 relies on the following ingredients:

- An MVBA protocol [34], an RBC protocol [19], and a threshold common-coin protocol (whose invocation is denoted by Coin.Get()), as elaborated in Appendix B.
- RobustRec protocol (and its batch version) to support multiplications between shared values or other necessary reconstructions, cf. Appendix E.
- A random oracle $H : \{0,1\}^* \to [L]$, three random oracles $\hat{H}, \tilde{H}, \check{H}$: $\{0,1\}^* \to \mathbb{Z}_p$, and another random oracle $H_{\mathsf{perm}} : \{0,1\}^* \to \mathbb{P}_N$ where $\mathbb{P}_N$ denotes the set of all $N!$ permutations over $[N]$.
- ComputePowers protocol, taking as input shares $[\![m]\!]$, a degree d, and preprocessing materials, to output secret shares $[\![m]\!]$, $[\![m^2]\!]$, $\dots$, $[\![m^d]\!]$, as Appendix F.1 describes.
- A local algorithm Solve, as in Appendix F.3, taking as input power sums $\{\sigma_k\}_{k\in[d]}$ where $\sigma_k = \sum_{i\in[d]} x_i^k$, to output the multiset of all roots $\{x_i\}_{i\in[d]}$ in a canonical order.

CONSTRUCTION. The pseudocode is presented in Algorithm 1, which proceeds in the following steps.

1. *Input submission (Line 1-5, for clients):* Each client robustly reconstructs $2L$ randomness prepared by the offline phase, denoted by $\{r_i^{(j)}\}_{i\in[2L]}$. It partitions them into two vectors, namely $(r_1^{(j)}, \dots, r_L^{(j)})$ and $(r_{L+1}^{(j)}, \dots, r_{2L}^{(j)})$. The client uniformly samples an index $i \in [L]$, adds 1 to the i-th entry of the first vector, and adds $m^{(j)}$ to the i-th entry of the second vector, thereby obtaining $\bar{I}^{(j)}$ and $\bar{M}^{(j)}$, respectively. The client then broadcasts the two masked vectors to servers via RBC.
2. *Input collection (Line 6-14, for servers):* Each server maintains a local set of clients whose RBC instances have completed. Upon receiving both $\bar{I}^{(j)}$ and $\bar{M}^{(j)}$ from client j, the server adds j to this set. Once the set reaches size N, the server invokes MVBA on its current snapshot. After MVBA terminates with a subset $\mathcal{L}$ of size N, each server locally unmasks the corresponding input shares. Specifically, for every $j \in \mathcal{L}$, the servers compute $[\![I_i^{(j)}]\!] = \bar{I}_i^{(j)} - [\![r_i^{(j)}]\!]$ and $[\![M_i^{(j)}]\!] = \bar{M}_i^{(j)} - [\![r_{i+L}^{(j)}]\!]$.

[5] Jumping ahead, we do not preprocess an equal number of shared powers for $k \in [2, N]$; smaller k is more frequently consumed, and too large k (e.g., close to N) happens only with a negligible probability (so its preprocessing can even be safely omitted). See Appendix G and I for analysis.

3. *Input validation (Line 15-25, for servers):* Since clients are not trusted, the servers validate each submission before aggregation. The validation enforces that $I^{(j)}$ is one-hot and that $M^{(j)}$ has its unique nonzero entry at the same index. Specifically, we follow the Schwartz–Zippel-style test from [4], the servers leverage a set of common randomness to combine $I^{(j)}$ and $M^{(j)}$ into a vector $w^{(j)}$, and check $\beta^{(j)} = 0$ to ensure that $w^{(j)}$ has Hamming weight at most one. Then, we also check $\alpha^{(j)} = 0$, which enforces $\sum_i I_i^{(j)} = 1$ and completes the one-hot check. The submission is accepted only if both checks pass.
4. *Optimistic mixing by random shifting then aggregating (Line 26-33, for servers):* For each valid input, the servers jointly sample a random offset and cyclically shift the shared vector so its non-zero entry is uniformly distributed over L entries. They then aggregate the shifted shared vectors and open the resulting counting vector $\mathbb{I}$ and message vector $\mathbb{T}$. For any k with $\mathbb{I}_k = 1$, the single message in entry k is delivered directly as $\mathbb{T}_k$.
5. *Algebraic mixing only for collided entries (Line 34-42, for servers):* For each index k with $\mathbb{I}_k \geq 2$, the entry contains a collision. The servers collect the shared k-th entries from all valid submissions and compute the corresponding power sums for degrees 2 through $\mathbb{I}_k$. After reconstructing these power sums, they derive the Newton basis polynomial and locally recover the colliding messages by solving the resulting system of equations.

When $L = cN$ for a constant c, each entry contains an expected constant number of messages, as all N messages are randomly placed among L positions. That said, compared with PowerMix, which solves an order-N algebraic-mixing for N messages, DumboMix1 only solves N instances of (expected) constant-order algebraic-mixing, thereby removing cubic scalar multiplications in PowerMix. A detailed complexity analysis is deferred to Appendix I.

Algorithm 1 DumboMix1

Input:
Public parameters: number of clients N, vector length L, and prime modulus p with $p > N$.
Client j's input: message $m^{(j)}$; secret shares $\{[\![r_i^{(j)}]\!]\}_{i\in[2L]}$ obtained in the preprocessing phase.
Server inputs: sufficient amount of shared randomness, Beaver triples, and shared powers.
Output: shuffled results msgs.

/* Executed by client j */
// Input submission
reconstructs $\{r_i^{(j)}\}_{i\in[2L]}$ from $\{[\![r_i^{(j)}]\!]\}_{i\in[2L]}$ using RobustRec
client j randomly selects an index $i \in [L]$
$\bar{I}^{(j)} \leftarrow [r_1^{(j)}, \ldots, r_{i-1}^{(j)}, 1 + r_i^{(j)}, r_{i+1}^{(j)}, \ldots, r_L^{(j)}]$
$\bar{M}^{(j)} \leftarrow [r_{L+1}^{(j)}, \ldots, r_{L+i-1}^{(j)}, m^{(j)} + r_{L+i}^{(j)}, r_{L+i+1}^{(j)}, \ldots, r_{2L}^{(j)}]$
RBC $\bar{I}^{(j)}$ and $\bar{M}^{(j)}$ to all servers

/* Executed by each server */
// Input collection
initialize $\mathcal{L}^* \leftarrow \varnothing$
upon receiving $I^{(j)}$ and $M^{(j)}$ from client j:
  $\mathcal{L}^* \leftarrow \mathcal{L}^* \cup \{j\}$
  if $|\mathcal{L}^*| = N$:
    invoke MVBA with a snapshot of $\mathcal{L}^*$ as input
wait until MVBA outputs $\mathcal{L}$, w.l.o.g., let $\mathcal{L} = \{1, \ldots, N\}$
for $i \in [L]$ and $j \in \mathcal{L}$
  $[\![I_i^{(j)}]\!] \leftarrow \bar{I}_i^{(j)} - [\![r_i^{(j)}]\!]$
  $[\![M_i^{(j)}]\!] \leftarrow \bar{M}_i^{(j)} - [\![r_{i+L}^{(j)}]\!]$
// Input validation
the servers jointly reconstruct a random coin $\leftarrow$ Coin.Get()
for $j \in \mathcal{L}$ **(in parallel)**
  for $i \in [L]$
    get $\hat{r}_i^{(j)}, \tilde{r}_i^{(j)}, \check{r}_i^{(j)}$ as $\hat{H}(\text{coin}||i||j), \tilde{H}(\text{coin}||i||j), \check{H}(\text{coin}||i||j)$
    compute $[\![\alpha^{(j)}]\!] = \sum_{i=1}^{L} [\![I_i^{(j)}]\!] - 1$
    compute $[\![w_i^{(j)}]\!] = [\![I_i^{(j)}]\!] \cdot \hat{r}_i^{(j)} + \tilde{r}_i^{(j)} \cdot [\![M_i^{(j)}]\!]$
    compute $[\![\bar{w}^{(j)}]\!] = \sum_{i=1}^{L} [\![w_i^{(j)}]\!]$
    $[\![\beta^{(j)}]\!] = (\sum_{i=1}^{L} [\![w_i^{(j)}]\!] \cdot \check{r}_i^{(j)})^2 - [\![\bar{w}^{(j)}]\!] \cdot \left(\sum_{i=1}^{L} [\![w_i^{(j)}]\!] \cdot (\check{r}_i^{(j)})^2\right)$
    open $\alpha^{(j)}$ and $\beta^{(j)}$
    if either of $\alpha^{(j)}$ and $\beta^{(j)}$ is nonzero
      set $\mathcal{L} \leftarrow \mathcal{L}/\{j\}$
//Optimistic mixing by random shifting then aggregating
for $j \in \mathcal{L}$ **(in parallel)**
  derive offset $s^{(j)}$ as $H(\text{coin}||j)$
  cyclically right-shift $[\![I^{(j)}]\!]$ and $[\![M^{(j)}]\!]$ by offset $s^{(j)}$
  update $[\![\mathbb{I}_i]\!] \leftarrow [\![\mathbb{I}_i]\!] + [\![I_i^{(j)}]\!]$
  update $[\![\mathbb{T}_i]\!] \leftarrow [\![\mathbb{T}_i]\!] + [\![M_i^{(j)}]\!]$
reconstruct $\mathbb{I}$ and $\mathbb{T}$, and initialize msgs $\leftarrow \varnothing$
for each entry $i \in [L]$:
  if $\mathbb{I}_i = 1$: append $\mathbb{T}_i$ to msgs
// Algebraic mixing only for collided entries
  else if $\mathbb{I}_i \geq 2$:
    set $d \leftarrow \mathbb{I}_i$
    for $j \in \mathcal{L}$: $\{[\![\sigma_k^{(j)}]\!]\}_{k\in[d]} \leftarrow \text{ComputePowers}([\![M_i^{(j)}]\!], d)$
    for $k \in [d]$: $[\![\sigma_k]\!] \leftarrow \sum_{j\in\mathcal{L}} [\![\sigma_k^{(j)}]\!]$
    $\{\tilde{m}_j\}_{j\in[d]} \leftarrow \text{Solve}(\{[\![\sigma_k]\!]\}_{k\in[d]})$
    append $\{\tilde{m}_j\}_{j\in[d]}$ to msgs
right-pad msgs with dummy symbols $\perp$ to length N
coin$' \leftarrow$ Coin.Get(), $\tilde{\pi} \leftarrow H_{\text{perm}}(\text{coin}')$, and msgs $\leftarrow \tilde{\pi}$(msgs)
remove all dummy symbols $\perp$ from msgs
return msgs

4.2 DumboMix2

Now we proceed with DumboMix2, which further improves DumboMix1 by reducing client-side overhead at the cost of additional server-side preprocessing. Note that DumboMix2 supports a message space $\mathbb{Z}_p^*$, as 0-messages submitted by clients will be rejected and not processed by servers.

PREPROCESSING MATERIALS. The preprocessing materials required by DumboMix1 are also used by DumboMix2. Unlike DumboMix1 where each client locally chooses a place-

ment of its message among an N-sized vector, DumboMix2 lets the servers collectively determine this random position. To support this methodology efficiently by shifting its cost to the offline phase, the servers additionally preprocess, for each client j: a shared random one-hot vector $[\![I^{(j)}]\!]$ and a shared random one-Hamming-weight vector $[\![U^{(j)}]\!]$, both with the same unique nonzero index. $[\![I^{(j)}]\!]$ can be efficiently preprocessed by Algorithm 2, borrowing the idea from demultiplexing protocols [56]; then $[\![U^{(j)}]\!]$ can be computed by MPC-multiplying each entry of $[\![I^{(j)}]\!]$ by a shared randomness.

Algorithm 2 Random One-Hot Vector Generation (Offline)

Input: Length L, where L is a power of 2; Secret-shared sign bits $\{[\![b_d]\!]\}_{d=0}^{\log_2 L-1}$, which can be preprocessed by Algorithm 12 in Appendix F.4 s.t. each b_d is randomly sampled from $\{-1,+1\}$.
Output: A secret-shared one-hot vector $[\![I]\!]$.

Procedure SplitNode($[\![x]\!],[\![b]\!]$):
1: $[\![c]\!] \leftarrow [\![x]\!] \cdot [\![b]\!]$
2: $[\![x_\ell]\!] \leftarrow 2^{-1} \cdot ([\![x]\!] - [\![c]\!])$
3: $[\![x_r]\!] \leftarrow 2^{-1} \cdot ([\![x]\!] + [\![c]\!])$
4: **return** $([\![x_\ell]\!],[\![x_r]\!])$

5: initialize $[\![T_0]\!],\ldots,[\![T_{2L-2}]\!]$ as 0s
6: $[\![T_0]\!] \leftarrow 1$
7: **for** $i = 0$ **to** $L-2$:
8: $d \leftarrow \lfloor \log_2(i+1) \rfloor$
9: $([\![T_{2i+1}]\!],[\![T_{2i+2}]\!]) \leftarrow$ SplitNode($[\![T_i]\!],[\![b_d]\!]$)
10: $[\![I]\!] \leftarrow ([\![T_{L-1}]\!],[\![T_L]\!],\ldots,[\![T_{2L-2}]\!])$
11: **return** $[\![I]\!]$

CONSTRUCTION. The core construction of DumboMix2 largely follows DumboMix1, except for how clients submit their messages to servers and how servers subsequently place each message at a random position in a vector. Its pseudocode is described in Algorithm 3, which proceeds in the next steps.

1. *Input submission (Line 1-5, for clients):* Each client first receives shares of a random value $u^{(j)}$ from the servers and reconstructs it locally. If $u^{(j)} = 0$, the client requests the servers to open $u^{(j)}$ to confirm the zero value, and then obtains a fresh mask. The clients then compute their masked inputs as $\bar{m}^{(j)} = m^{(j)} \cdot (u^{(j)})^{-1}$ and broadcast the masked values to the servers via RBC.

2. *Input collection (Line 6-13, for servers):* Similarly, the servers invoke MVBA to agree on a set of N completed submissions. After MVBA terminates and outputs a set $\mathcal{L}$, each server locally computes, for every client $j \in \mathcal{L}$ and index $i \in [L]$, $[\![M_i^{(j)}]\!] = \bar{m}^{(j)} \cdot [\![U_i^{(j)}]\!]$.

3. *Optimistic mixing by aggregating (Line 14-18, for servers):* Clients now submit only masked messages, allowing the servers to directly aggregate the extended vectors without input validation or cyclic shifting.

4. *Algebraic mixing only for collided entries (Line 18, i.e. Lines 34–42 of Algorithm 1):* As in DumboMix1, the servers invoke algebraic mixing only on entries containing collisions, which suffices to recover all messages.

Algorithm 3 DumboMix2

Input:
Public parameters: number of clients N, vector length L, and prime modulus p with $p > N$.
Client j's input: message $m^{(j)}$; secret shares $[\![u^{(j)}]\!]$ obtained in the preprocessing phase, where $[\![u^{(j)}]\!] = \sum_{i\in[L]} [\![U_i^{(j)}]\!]$.
Server inputs: A set of shared random one-hot vectors $\{[\![I^{(j)}]\!]\}$ and the corresponding shared random one-Hamming-weight vector $\{[\![U^{(j)}]\!]$ for each client; sufficient amount of shared randomness and shared powers.
Output: shuffled results msgs.

/* Executed by client j */
1: reconstruct $[\![u^{(j)}]\!]$ via RobustRec
2: **if** $u^{(j)} = 0$
3: discard it and request a fresh randomness $u^{(j)}$
4: set $\bar{m}^{(j)} = m^{(j)} \cdot (u^{(j)})^{-1}$
5: RBC $\bar{m}^{(j)}$ to all servers

/* Executed by each server */
6: initialize $\mathcal{L}^* \leftarrow \varnothing$
7: upon receiving non-zero $\bar{m}^{(j)}$ from client j:
8: $\mathcal{L}^* \leftarrow \mathcal{L}^* \cup \{j\}$
9: **if** $|\mathcal{L}^*| = N$:
10: invoke MVBA with a snapshot of $\mathcal{L}^*$ as input
11: wait until MVBA outputs $\mathcal{L}$, w.l.o.g., let $\mathcal{L} = \{1,\ldots,N\}$
12: **for** each $j \in \mathcal{L}$ and $i \in [L]$
13: set $[\![M_i^{(j)}]\!] = \bar{m}^{(j)} \cdot [\![U_i^j]\!]$
//Optimistic mixing by aggregating
14: **for** $j \in \mathcal{L}$ **(in parallel)**
15: update $[\![\mathbb{I}_i]\!] \leftarrow [\![\mathbb{I}_i]\!] + [\![I_i^{(j)}]\!]$, and update $[\![\mathbb{T}_i]\!] \leftarrow [\![\mathbb{T}_i]\!] + [\![M_i^{(j)}]\!]$
16: reconstruct $\mathbb{I}$ and $\mathbb{T}$, and initialize msgs $\leftarrow \varnothing$
17: **for each** entry $i \in [L]$:
18: **if** $\mathbb{I}_i = 1$: append $\mathbb{T}_i$ to msgs
19: /- - **Same procedure as in Lines 34-42 of Algorithm 1** - -/
20: **return** msgs

4.3 Optimized Variants and Future Direction

Server-optimized DumboMix1* and DumboMix2*. We now describe two practical variants that improve the concrete efficiency of DumboMix1 and DumboMix2, respectively, by trading a modest increase in client-side communication for reduced server-side computation, as inspired by Blinder [4].

Recall that when multiple messages are randomly placed at the same entry, the servers invoke algebraic mixing to recover these colliding messages. A key observation is that most collision entries contain only two messages in typical parameter regimes. Under the standard balls-into-bins model, when N messages are placed uniformly into L entries, the occupancy of each entry is well approximated by a Poisson random variable with mean $\lambda = N/L$. For example, under Blinder's recommended parameter choice $L = 2.75N$, conditioned on an entry

being non-singleton, the probability that it contains exactly two messages is $\frac{\Pr[X=2]}{\Pr[X\geq 2]} = \frac{e^{-\lambda}\lambda^2/2}{1-e^{-\lambda}(1+\lambda)} \approx 82\%$.

Thus, a large fraction of collision entries can be handled by optimizing the two-message case. Inspired by Blinder [4], we let each client submit both its message and the message's square. These two values allow servers to resolve two-message collisions directly, avoiding an invocation of ComputePowers, which is the (expected) dominating computation in algebraic mixing. The additional submitted value requires only an extra input-validation check enforcing the square relation. We denote the resulting optimized variants by DumboMix1* and DumboMix2*, and defer their full descriptions with this intuitive optimization to Appendix H.

Why *not* further "improve" DumboMix2 **with involved preprocessing?** A seemingly direct improvement to DumboMix2 is to preprocess a uniformly random secret-shared permutation matrix, i.e., an $N \times N$ binary matrix with exactly one 1 in each row and column. Each row can then replace the one-hot vector used in DumboMix2, eliminating collisions and hence the need for fallback algebraic mixing in the online phase.

The remaining challenge is that robustly preprocessing such a matrix could be prohibitively expensive. In a folklore approach, at least $t+1$ servers have to each sample a random permutation matrix, which are then multiplied to obtain the final uniformly random permutation matrix [57]. This incurs $O(tN^3)$ MPC multiplications per preprocessing a batch of N messages. Adapting the idea of [39] can reduce this cost to $O(\min(\kappa,t)N^3)$, where κ is a statistical security parameter typically on the order of tens, but the cubic MPC multiplications remain impractical for large N. In contrast, the current preprocessing of DumboMix2 requires only $O(N^2)$ MPC multiplications per N-message batch, and DumboMix1 requires even fewer. Thus, our designs of DumboMix strike a balance between the critical efficiency of online-phase mixing and affordable preprocessing costs. Appendix G provides a detailed analysis for offline preprocessing complexities.

We therefore remark that any more efficient method for robustly preprocessing shared random permutation matrices can be directly plugged into DumboMix2 to further improve performance, as this avoids fallback to algebraic mixing. Nevertheless, developing more efficient techniques for this involved preprocessing task remains an interesting open problem.

4.4 Analysis of Complexity and Security

Computation complexity. Here, we count scalar multiplications, MPC multiplications, and reconstructions, which dominate the cryptographic cost. For each input, validation costs $O(L)$ scalar multiplications, together with $O(1)$ MPC multiplications and reconstructions. Thus, validating N inputs costs $O(NL)$. After aggregation, the servers open an L-entry vector and resolve only collided entries. By Lemma 1, the expected cost of constructing the Newton-basis polynomials over all collisions is $O\left(N^2+\frac{N^3}{L}\right)$, and root finding costs $O(N^2)$ field operations. Hence, the overall expected computation is $O\left(NL+N^2+\frac{N^3}{L}\right)$. Setting $L=\Omega(N)$ balances the L-dependent terms and yields $O(N^2)$ expected computation.

Communication complexity. During input collection, each server participates in $O(N)$ RBC instances. With $L = cN$, this takes $\tilde{O}(N^2)$ communication for DumboMix1 and DumboMix1*, where clients submit length-L vectors, and $\tilde{O}(N)$ communication for DumboMix2 and DumboMix2*, where clients submit constant-size inputs. The MVBA step costs $\tilde{O}(N)$. Validation uses only a constant number of MPC multiplications and openings per input, while opening the aggregated vector and resolving collisions contribute at most $\tilde{O}(N^2)$ communication overall. Thus, the total communication overhead of DumboMix is $\tilde{O}(N^2)$.

Security analysis. Our protocols securely realize the ideal functionality $\mathcal{F}_{\mathsf{AAB}}$. Informally, all variants of DumboMix provide the following security guarantees for robust asynchronous shuffling, as captured by $\mathcal{F}_{\mathsf{AAB}}$:

- *Robustness.* All honest servers will eventually terminate the invocation and output an agreed message sequence.
- *Optimal delivery ratio.* In each invocation, the protocol guarantees that at least $N-F$ honest inputs are delivered.
- *Privacy of undelivered messages.* Every message submitted by an honest client is either delivered or remains private from the adversary.
- *Unlinkability.* For every output generated by an honest client, the advantage of any PPT adversary correctly linking that output to its sender is at most $\frac{1}{N-F}$.

Proof sketch. We outline the main ideas underlying the security proofs below and defer the full simulation-based proof for each variant of DumboMix to Appendix J.

For *robustness*: our AAB protocols satisfy termination and agreement. Specifically, MVBA ensures that all honest servers terminate with the same set $\mathcal{L}$. Given correctly generated Beaver triples and shared randomness, the random-coin protocol consistently reconstructs the required randomness, while the MPC multiplications terminate and yield consistent results across honest servers. The remaining steps, including aggregation, power-sum computation, and root finding, involve only deterministic local computations, with the required values that are correctly reconstructed to all servers when $t < n/3$. Hence, all honest servers eventually terminate and output the same message sequence.

For *optimal delivery ratio*: MVBA outputs an N-client set $\mathcal{L}$, containing at least $N-F$ honest clients. Input validation accepts all well-formed inputs (including all honest clients in $\mathcal{L}$), and rejects malformed ones (which must be a subset of malicious inputs). Each accepted input is placed into exactly one vector entry. Input validation and correctness of MPC addition

ensure the aggregated counting vector correctly records the number of messages in each entry of the aggregated message vector. Singleton entries are opened directly, while collided entries are recovered from power sums via Newton identities. Thus, at least $N - F$ honest messages are correctly delivered.

For *privacy of undelivered messages*: any selected honest input is delivered by the previous argument. Hence, an undelivered honest input must lie outside the set $\mathcal{L}$, never enter validation or mixing, and remain protected by secret sharing.

For *unlinkability*, each honest client places its message uniformly at random in the vector, making its position independent of the sender. A message in a singleton entry is recovered directly, while messages in a collided entry are recovered from the aggregate power sums in a canonical order that is independent of their senders. Hence, the transcript reveals no linkage between honest messages and their senders.

5 Implementation and Evaluations

We implement the different variants of DumboMix and evaluate them with RabbitMix [24], Blinder [4], a butterfly-style switching network [58], and PowerMix [58], all within the same robust AMPC framework. RabbitMix and Blinder have no asynchronous implementation before. As a byproduct, we also implement the robust asynchronous versions of them.

Online implementation. We implement DumboMix atop the AMPC online-phase implementation hbMPC [58]. We also implement RabbitMix and Blinder-1 (as Blinder-2's input compaction is inefficient to instantiate in our setting). RabbitMix is implemented using a mixed-radix representation with base $\sqrt{N}$. For fairness, we use the same parameter setting $L = 2.75 * N$, as Blinder suggests. For consistency, RabbitMix, PowerMix, and DumboMix all adopt the same root-finding implementation from PowerMix [58].

Our performance-optimized offline-phase implementation. To support the robust distributed generation of preprocessing materials required by all these AAB protocols, we further improve DumboMPC to obtain a computationally more efficient offline-phase implementation, DumboMPC++. As a by-product, DumboMPC++ realizes a suite of performance-optimized preprocessing protocols for generating shared randomness and multiplication triples. The implementation improves concrete computational efficiency by eliminating reliance on pairings over pairing-friendly elliptic curves and by incorporating a more efficient batch-verification technique for zero-knowledge proofs of product relations over cryptographic commitments (cf. Appendix D for details of the offline phase). For fair comparison, DumboMPC++ forked DumboMPC, using the same programming language and identical underlying libraries: point-to-point communication is implemented using unauthenticated TCP sockets, messages are serialized in JSON, and concurrency is managed through Python's `asyncio` framework. We also use the same cryptographic libraries and instantiations, namely the BLS12-381 elliptic curve together with the `gnark-crypto` library [12] for generalized Pedersen commitments. [6]

Experimental setup. All experiments are conducted in Docker containers on Alibaba Cloud ECS `c9ae.8xlarge` instances. Each server runs on a dedicated ECS instance, and all instances have the same hardware configuration: 32 vCPUs and 64 GB of RAM. We evaluate the systems with $n \in \{4, 7, 10, 16, 22, 31\}$ servers. In the LAN setting, all instances are deployed in the same ECS region (Ulanqab). To emulate WAN conditions, we use Linux `tc` to cap the uplink bandwidth of each instance at 500 Mbps and to impose a round-trip time (RTT) of 150 ms.

5.1 Evaluations on Robust AABs

This subsection focuses on evaluating the critical end-to-end online performance of robust AABs (i.e., from when the clients provide messages until the mixing is completed).

Effects of optimizations in Section 4.3. Figure 3 shows that, with Blinder-style submission (each client additionally submits their message square), DumboMix1* and DumboMix2* are roughly 40% faster than DumboMix1 and DumboMix2, respectively. We defer additional results to Appendix K.1.

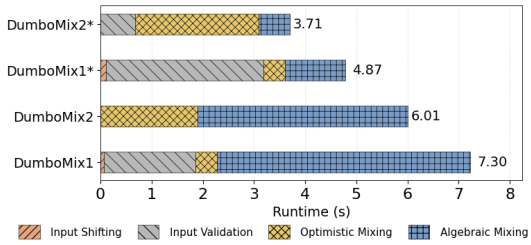


Figure 3: Breakdown of the online-phase runtime for each DumboMix variant (WAN, $n = 10$, $N = 1024$)

Throughout the paper, we will experimentally compare DumboMix1*/DumboMix2* with existing robust AAB approaches, including the AMPC-based SwitchNet [58], PowerMix [58], and the AMPC variant of RabbitMix [24]. We evaluate these baselines in their original forms, as our optimization developed for DumboMix does not apply to them. Appendix K.3 further compares DumboMix with an AMPC implementation of Blinder [4], a non-robust DC-net-based design that is vulnerable to "re-attempt linkability" under asynchrony, to reveal that DumboMix provides guaranteed delivery "almost for free."

Runtime comparison. Table 2 summarizes the end-to-end running time of all protocols for different numbers of clients

[6] Although the pairing-friendly BLS12-381 elliptic curve is not necessarily the optimal choice for implementing our protocols, given that our design no longer relies on pairings, we use it throughout the evaluation to ensure that all protocols operate over the same message space and the identical secret-sharing field, thereby facilitating a fair and consistent comparison.

Table 2: Runtime comparison between robust AAB implementations under varying N (with fixed servers $n = 31$).

N	SwitchNet (sec)		PowerMix (sec)		RabbitMix (sec)		DumboMix1* (sec, speedup)		DumboMix2* (sec, speedup)	
	LAN	WAN	LAN	WAN	LAN	WAN	LAN	WAN	LAN	WAN
2^6	0.44	5.94	0.92	3.18	0.75	1.47	0.13 ($\geq$ 3.4×)	1.24 ($\geq$ 1.2×)	0.16 ($\geq$ 2.8×)	0.90 ($\geq$ 1.6×)
2^7	0.87	8.31	1.40	3.63	2.96	4.34	0.22 ($\geq$ 3.9×)	1.34 ($\geq$ 2.7×)	0.20 ($\geq$ 4.3×)	0.94 ($\geq$ 3.9×)
2^8	1.85	11.51	2.63	4.79	10.94	13.77	0.48 ($\geq$ 3.8×)	1.57 ($\geq$ 3.1×)	0.34 ($\geq$ 5.5×)	1.15 ($\geq$ 4.2×)
2^9	4.25	16.41	5.86	7.91	44.09	50.67	1.23 ($\geq$ 3.5×)	2.39 ($\geq$ 3.3×)	0.75 ($\geq$ 5.7×)	1.62 ($\geq$ 4.9×)
2^{10}	9.93	24.85	17.54	19.07	162.15	179.80	3.62 ($\geq$ 2.7×)	4.84 ($\geq$ 3.9×)	2.46 ($\geq$ 4.0×)	3.41 ($\geq$ 5.6×)
2^{11}	23.89	41.65	76.44	78.06	690.73	738.06	14.11 ($\geq$ 1.7×)	15.67 ($\geq$ 2.7×)	8.95 ($\geq$ 2.7×)	9.67 ($\geq$ 4.3×)

in LAN/WAN with $n = 31$ servers. Both DumboMix1* and DumboMix2* constantly outperform prior robust AAB protocols. In LAN, DumboMix1* achieves up to 3.9× speedup over the best baseline, while DumboMix2* achieves up to 5.7× speedup. In WAN, our performance gain becomes even more pronounced: DumboMix1* improves performance by up to 3.9×, and DumboMix2* by up to 5.6×. For example, at $N = 2^{10}$, DumboMix2* completes in 3.41 sec in WAN, whereas SwitchNet, PowerMix, and RabbitMix require 24.85, 19.07, and 179.80 sec, respectively. Additional results for varying server numbers are reported in Appendix K.

Communication comparison. We then report the communication overhead of the evaluated AAB protocols.

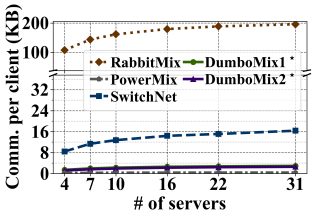


Figure 4: Communication vs. servers number n ($N = 512$)

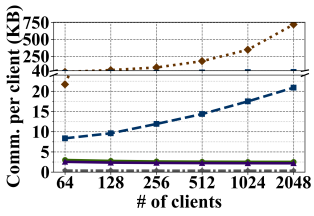


Figure 5: Communication vs. clients number N ($n = 16$)

Fig. 4 shows the communication when the number of clients N is fixed to 512 and the number n of servers varies over $n \in \{4, 7, 10, 16, 22, 31\}$. The results show that the communication of our constructions grows slowly with the number of servers and remains low across all tested settings, and substantially outperforms PowerMix and SwitchNet, as RabbitMix and SwitchNet require $O(N^2)$ and $O(N \log^2 N)$ MPC multiplications, respectively. Though PowerMix presents the lowest communication (as no use of MPC multiplications), DumboMix1* and DumboMix2* also maintain modest communication overhead, requiring 2.8 KB and 2.5 KB per server per input, respectively. Fig. 5 further reports the communication when the number of servers is fixed to $n = 16$, and the number N of clients varies. As N increases, the communication cost of RabbitMix and SwitchNet increases dynamically, again due to their heavy use of online MPC multiplications. By contrast, both DumboMix1* and DumboMix2* exhibit slower growth of communication overhead in N (due to less use of MPC multiplications), indicating more scalable performance for larger anonymity sets with more clients.

5.2 Affordability of Preprocessing

Optimized offline-phase framework. For affordable preprocessing, we first augment DumboMPC, the state-of-the-art robust offline implementation for AMPC, with our optimized randomness- and triple-generation protocols (cf. Appendix D for their standalone performance).

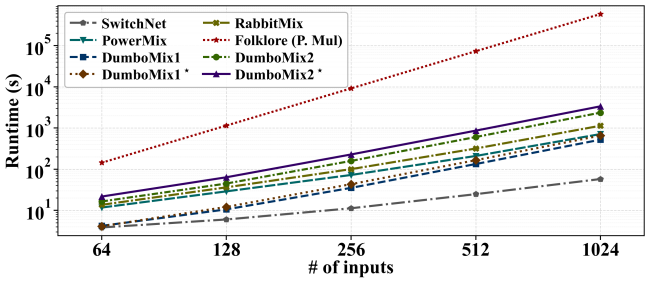


Figure 6: Preprocessing runtime vs. batch size N of inputs, compared with existing AABs (WAN with $n = 4$ servers)

Preprocessing performance. Prior AAB schemes do not provide asynchronous preprocessing implementations; as a byproduct, we implement their preprocessing circuits within DumboMPC++ using algorithms in Appendix F. We also evaluate an optimized folklore preprocessing approach [68] that generates a secret-shared random permutation matrix. As discussed in Section 4.3, preparing such a matrix more efficiently beyond this folklore idea remains an open problem.

As shown in Figure 6, the folklore approach is, as expected, prohibitively expensive, taking about 20 hours at $N = 512$. Among the remaining constructions, SwitchNet [58] achieves the lowest preprocessing cost, followed by DumboMix1 and DumboMix1* at approximately 2 and 2.5 minutes, respectively. Although DumboMix2 and DumboMix2* are more expensive than RabbitMix [24], their preprocessing times remain affordable, at about 10 and 14 minutes at this scale, respectively. All preprocessing materials are input-independent, so servers can prepare them in advance and continuously replenish a pool for future online executions. As long as sufficient preprocessing materials are available, preprocessing remains off the critical online path and thus has little impact on the end-to-end latency perceived by users. [7]

[7] The preprocessing tasks of multiple AAB instances can also be parallelized by allocating additional computation and bandwidth, allowing the offline replenishment rate to more closely match the online consumption rate.

6 Conclusion

We present DumboMix, an efficient framework for anonymous broadcast that provides guaranteed output delivery and optimal resilience in the fully asynchronous setting. Compared with existing mixing methods suitable/adaptable for the same setting (i.e., RabbitMix [24], PowerMix [58], and the butterfly switching network [58]), our new mixing methods, DumboMix1 and DumboMix2 (as well as their server-side optimized variants DumboMix1* and DumboMix2*), asymptotically improve at least one of the following critical online-phase complexity measures: the number of rounds, MPC multiplications, and scalar multiplications. These asymptotic improvements translate into significant speedups in both LAN and WAN environments, while shuffling up to 2048 messages.

References

[1] Ittai Abraham, Gilad Asharov, Shravani Patil, and Arpita Patra. Perfect asynchronous mpc with linear communication overhead. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 280–309. Springer, 2024.

[2] Ittai Abraham, Gilad Asharov, Arpita Patra, and Gilad Stern. Perfectly secure asynchronous agreement on a core set in constant expected time. *Cryptology ePrint Archive*, 2023.

[3] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically optimal validated asynchronous byzantine agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

[4] Ittai Abraham, Benny Pinkas, and Avishay Yanai. Blinder–scalable, robust anonymous committed broadcast. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1233–1252, 2020.

[5] Shashank Agrawal, Chaya Ganesh, and Payman Mohassel. Non-interactive zero-knowledge proofs for composite statements. In *Annual International Cryptology Conference*, pages 643–673. Springer, 2018.

[6] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. {MCMix}: anonymous messaging via secure multiparty computation. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1217–1234, 2017.

[7] Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Balanced byzantine reliable broadcast with near-optimal communication and improved computation. In *Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing*, pages 399–417, 2022.

[8] Akhil Bandarupalli, Xiaoyu Ji, Aniket Kate, Chen-Da Liu-Zhang, Daniel Pöllmann, and Yifan Song. Velox: Scalable fair asynchronous mpc from lightweight cryptography. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, pages 1799–1813, 2025.

[9] Paulo SLM Barreto, Ben Lynn, and Michael Scott. Constructing elliptic curves with prescribed embedding degrees. In *International conference on security in communication networks*, pages 257–267. Springer, 2002.

[10] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Advances in Cryptology–CRYPTO'91: Proceedings 11*, pages 420–432, 1992.

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

[12] Gautam Botrel, Thomas Piellard, Youssef El Housni, Arya Tabaie, Gus Gutoski, and Ivo Kubjas. Consensys/gnark-crypto: v0.11.2, 2023.

[13] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 1292–1303, 2016.

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

[15] Benedikt Bünz, Shashank Agrawal, Mahdi Zamani, and Dan Boneh. Zether: Towards privacy in a smart contract world. In *International Conference on Financial Cryptography and Data Security*, pages 423–443. Springer, 2020.

[16] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[17] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and efficient asynchronous broadcast protocols. In *Advances in Cryptology-CRYPTO 2001: 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19–23, 2001 Proceedings*, pages 524–541. Springer, 2001.

[18] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 2005.

[19] Christian Cachin and Stefano Tessaro. Asynchronous verifiable information dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 191–201. IEEE, 2005.

[20] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In *International conference on the theory and application of cryptology and information security*, pages 342–372. Springer, 2020.

[21] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology*, 1(1):65–75, 1988.

[22] David Chaum, Debajyoti Das, Farid Javani, Aniket Kate, Anna Krasnova, Joeri De Ruiter, and Alan T Sherman. cmix: Mixing with minimal real-time asymmetric cryptographic operations. In *International conference on applied cryptography and network security*, pages 557–578. Springer, 2017.

[23] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[24] Chongwon Cho, Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. {Rabbit-Mix}: Robust algebraic anonymous broadcast from additive bases. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3151–3168, 2024.

[25] David Cole. We kill people based on metadata. The New York Review of Books, 2014. accessed 2026-05-04. URL: https://www.nybooks.com/articles/2014/05/10/we-kill-people-based-metadata.

[26] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*, pages 321–338. IEEE, 2015.

[27] Henry Corrigan-Gibbs and Bryan Ford. Dissent: accountable anonymous group messaging. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 340–350, 2010.

[28] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. Proactively accountable anonymous messaging in verdict. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 147–162, 2013.

[29] Ronald Cramer, Ivan Damgård, and Yuval Ishai. Share conversion, pseudorandom secret-sharing and applications to secure computation. In *Theory of Cryptography Conference*, pages 342–362. Springer, 2005.

[30] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference*, pages 572–590. Springer, 2007.

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

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

[33] Wentao Dong, Peipei Jiang, Huayi Duan, Cong Wang, Lingchen Zhao, and Qian Wang. Ring of gyges: Accountable anonymous broadcast via secret-shared shuffle. In *Proc. of NDSS*, 2025.

[34] Sisi Duan, Xin Wang, and Haibin Zhang. Fin: Practical signature-free asynchronous common subset in constant time. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 815–829, 2023.

[35] Sisi Duan, Xin Wang, and Haibin Zhang. Practical signature-free asynchronous common subset in constant time. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[36] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In Ernest F. Brickell, editor, *Advances in Cryptology — CRYPTO' 92*, pages 139–147, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[37] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. In *Proc. of NDSS*, 2022.

[38] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, and Dan Boneh. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1775–1792, 2021.

[39] Hanwen Feng, Yingzi Gao, Yuan Lu, Qiang Tang, and Jing Xu. Practical asynchronous distributed key reconfiguration and its applications. In *2026 IEEE Symposium on Security and Privacy (SP)*, pages 1253–1272. IEEE, 2026.

[40] Lee Ferran. Ex-nsa chief: "we kill people based on metadata". ABC News, 2014. accessed 2026-05-04. URL: https://abcnews.com/blogs/headlines/2014/05/ex-nsa-chief-we-kill-people-based-on-metadata.

[41] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 640–658. Springer, 2014.

[42] Vipul Goyal, Yanyi Liu, and Yifan Song. Communication-efficient unconditional mpc with guaranteed output delivery. In *Annual International Cryptology Conference*, pages 85–114. Springer, 2019.

[43] Vipul Goyal, Chen-Da Liu-Zhang, and Yifan Song. Towards achieving asynchronous mpc with linear communication and optimal resilience. In *Annual International Cryptology Conference*, pages 170–206. Springer, 2024.

[44] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed output delivery comes free in honest majority mpc. In *Annual International Cryptology Conference*, pages 618–646. Springer, 2020.

[45] Jens Groth and Victor Shoup. Design and analysis of a distributed ecdsa signing service. *Cryptology ePrint Archive*, 2022.

[46] Bingyong Guo, Yuan Lu, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. Speeding dumbo: Pushing asynchronous bft closer to practice. In *29th Annual Network and Distributed System Security Symposium (NDSS 2022)*. The Internet Society, 2022.

[47] Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In *Annual International Cryptology Conference*, pages 463–482. Springer, 2006.

[48] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification, 2024. Accessed: 2026-05-05. URL: https://zips.z.cash/protocol/protocol.pdf.

[49] Margaret Hu. Metadeath: How does metadata surveillance inform lethal consequences? In *Privacy and power: A transatlantic dialogue in the shadow of the NSA-affair*, pages 229–256. Cambridge University Press, 2017.

[50] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology - ASIACRYPT 2010*, pages 177–194, 2010.

[51] Marcel Keller. Mp-spdz: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, pages 1575–1590, 2020.

[52] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, Somya Sangal, et al. Ruffle: Rapid 3-party shuffle protocols. *Proceedings on Privacy Enhancing Technologies*, 2023.

[53] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. *Proceedings on Privacy Enhancing Technologies*, 2016.

[54] Albert Kwon, David Lu, and Srinivas Devadas. {XRD}: Scalable messaging system with cryptographic privacy. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 759–776, 2020.

[55] Simon Langowski, Sacha Servan-Schreiber, and Srinivas Devadas. Trellis: Robust and scalable metadata-private anonymous broadcast. In *Proc. of NDSS*, 2022.

[56] John Launchbury, Iavor S Diatchki, Thomas DuBuisson, and Andy Adams-Moran. Efficient lookup-table protocol in secure multiparty computation. In *Proceedings of the 17th ACM SIGPLAN international conference on Functional programming*, pages 189–200, 2012.

[57] Donghang Lu and Aniket Kate. Rpm: Robust anonymity at scale. *Proceedings on Privacy Enhancing Technologies*, 2023.

[58] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronous mpc and its application to anonymous communication. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 887–903, 2019.

[59] Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *Proceedings of the 39th symposium on principles of distributed computing*, pages 129–138, 2020.

[60] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 31–42, 2016.

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

[62] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. Coinshuffle: Practical decentralized coin mixing for bitcoin. In *European symposium on research in computer security*, pages 345–364. Springer, 2014.

[63] Tim Ruffing, Pedro Moreno-Sanchez, and Aniket Kate. P2p mixing and unlinkable bitcoin transactions. In *Proc. of NDSS*, 2017.

[64] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE symposium on security and privacy*, pages 459–474. IEEE, 2014.

[65] Jacob T Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)*, 27(4):701–717, 1980.

[66] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[67] Yuan Su, Yuan Lu, Jiliang Li, Yuyi Wang, Chengyi Dong, and Qiang Tang. Dumbo-mpc: efficient fully asynchronous mpc with optimal resilience. In *USENIX Security*, volume 2024, 2025.

[68] Wei Tang, Yuan Lu, Chao Jin, Jiliang Li, and Khin Mi Mi Aung. Single-round robust secret-shared shuffling with application to deployable shuffled differential privacy. *IEEE Transactions on Dependable and Secure Computing*, 2026. Accepted for publication.

[69] Yurek Thomas, Luo Licheng, Fairoze Jaiden, Kate Aniket, and Miller Andrew. hbacss: How to robustly share many secrets. In *Proceedings of the Network and Distributed Systems Security Symposium 2022*, NDSS '22, pages 1187–1201, 2022.

[70] Voltaire. Performance benchmarks. https://voltaire.tevm.sh/performance. Accessed: 2026-05-07.

[71] Avi Wigderson, MB Or, and S Goldwasser. Completeness theorems for noncryptographic fault-tolerant distributed computations. In *Proceedings of the 20th Annual Symposium on the Theory of Computing (STOC'88)*, pages 1–10, 1988.

[72] Xinxin Xing, Yizhong Liu, Boyang Liao, Jianwei Liu, Bin Hu, Xun Lin, Yuan Lu, and Tianwei Zhang. Gossamer: Lightweight and linear-communication asynchronous (dynamic proactive) secret sharing and the applications. *Cryptology ePrint Archive*, 2025.

[73] Lei Yang, Seo Jin Park, Mohammad Alizadeh, Sreeram Kannan, and David Tse. {DispersedLedger}:{High-Throughput} byzantine consensus on variable bandwidth networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 493–512, 2022.

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

[75] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*, pages 216–226. Springer, 1979.

A Additional Discussion on Related Work

Key online performance issues: Non-constant rounds, cubic local operations, or quadratic MPC mul.. As Table 1 shows, all existing mixing circuits suitable for AMPC frameworks suffer from high complexities, the most critical performance bottleneck in constructing practical robust AAB protocols. Particularly, the idea of evaluating a mix-net (e.g., the butterfly switching network) within AMPC has a multiplicative depth of $O(\log^2 N)$ for shuffling N messages [58], directly causing $O(\log^2 N)$ online rounds (which is costly in WAN, i.e., when $n = 1024$ and RTT=100ms, $\log^2 N$ rounds already imply a latency of 10 seconds).

On the other hand, PowerMix [58] adopts an algebraic shuffling approach to achieve a single online round, but results in asymptotically costly $N^3/2$ scalar multiplications between shared and public values. RabbitMix [24] further eliminates the cubic term of scalar multiplications and preserves $O(1)$ online rounds, but incurs $O(N^2)$ MPC multiplications between shared values. Although the quadratic MPC multiplications in RabbitMix appear asymptotically more appealing than the cubic local overhead in PowerMix at first glance, both of them can be concretely prohibitive at typical system scales, such as several thousand clients. As Fig. 1 illustrates, when shuffling 2048 messages in hbMPC [58] with a small set of 4 servers, $O(N^2)$ MPC multiplications in RabbitMix and $O(N^3)$ scalar multiplications in PowerMix become both prohibitive, with RabbitMix even at least 5× slower than PowerMix.

Another work Blinder [4] improves efficiency by relaxing the ideal functionality of not robustly shuffling all input messages, leading to the threat of *re-attempt linkability*, as aforementioned and discussed in detail in Appendix C.

Offline performance issues: High computational overhead in prior robust implementations. Prior AMPC implementations [45, 58, 67] typically adopt an offline–online paradigm to enable a relatively fast online phase that is (almost) information-theoretic (IT) and relies only on plain

t-degree Shamir secret sharing [66]. However, experimental evidence also reveals that the robust offline protocol in AMPC (which runs distributed protocols to preprocess secret shares of randomness and Beaver multiplication triples [10] for the online phase) is substantially slower [45, 58, 67]. In particular, when there are n=31 servers, the preprocessing throughput of the state-of-the-art robust offline implementation DumboMPC is even lower than 100 triples per second, corresponding to another performance bottleneck of constructing practical AAB.

The performance bottleneck in DumboMPC is dominated by its large computational overhead, as it relies heavily on the *compact* KZG polynomial commitment [50] (along with a large batch of zk-proofs for product relationships over these commitments) to simultaneously realize robustness, optimal resilience, and communication efficiency. Nevertheless, KZG in turn requires pairing-friendly elliptic curves (ECs), e.g., BLS12 [9], whose basic group operations are several times slower than on standard ECs without pairings (e.g., secp256k1), let alone even more costly pairings [70], leading to significantly higher computing cost on such ECs (e.g., for verifications of zk-proof and commitment). On the other hand, the alternative robust triple-generation protocol GS23 [45] (using only pairing-free ECs by Pedersen polynomial commitments [61] instead of KZG) incurs asymptotically higher per-triple overhead in the presence of Byzantine participants.

B Deferred Details on Preliminaries

Multi-valued validated asynchronous Byzantine agreement (MVBA) [3, 17, 46, 59] extends binary asynchronous Byzantine agreement to values from a general domain $\mathcal{X}$. In an MVBA protocol executed among n parties, each party proposes a value in $\mathcal{X}$, and all honest parties agree on an output $v \in \mathcal{X}$ satisfying a predefined validation predicate Q. The predicate may also depend on a party's local state [2, 31, 35, 74], i.e., $Q : \mathcal{X} \times \mathcal{S} \to \{0,1\}$, where $\mathcal{S}$ denotes the state space. We assume that Q is monotone and eventually unanimous: once an honest party validates a value, the value remains valid for that party, and every honest party can eventually validate it.

An MVBA protocol tolerating up to t Byzantine corruptions satisfies the following properties, except with negligible probability.

- *Termination.* If every honest party invokes the protocol with a valid input, then every honest party eventually outputs.
- *Agreement.* any two honest nodes output v_i and v_j, respectively, then $v_i = v_j$;
- *External validity.* If an honest party outputs v, then v satisfies the validation predicate Q.

Instantiation. For a fair comparison with DumboMPC [67], we use the same MVBA instantiation in our implementation. Notably, DumboMPC introduces a deterministic fast path to MVBAs, so the resulting instantiation can terminate within five rounds, if the network is synchronous and the designated leading party is also honest. In the worst case, each invocation on an n-bit input incurs $O(\kappa n^3)$ bits of communication and still probabilistically terminates in *expected* constant asynchronous rounds.

Reliable broadcast (RBC) [7, 14, 19, 32, 60] is an asynchronous broadcast primitive that allows a designated sender to disseminate a message consistently to all honest parties, even in the presence of Byzantine faults. An RBC protocol executed among n parties satisfies the following properties, except with negligible probability.

- *Agreement.* If two honest parties deliver messages m and m', respectively, then $m = m'$.
- *Validity.* If the sender is honest and broadcasts m, then every honest party eventually delivers m.
- *Totality.* If any honest party delivers a message m, then every honest party eventually delivers m.

Instantiation. For broadcasting the large message, we instantiate RBC from [19, 60], which uses erasure coding and Merkle-tree authentication to achieve $O(n|v| + \kappa n^2 \log n)$ communication for a sufficiently large input v (e.g., $|v| > \kappa n$).

Threshold common coin is an asynchronous distributed protocol that allows the participating parties to obtain a common unpredictable value for a given public context ctx. In an (n,t)-threshold common coin protocol, the honest parties eventually output a common value $r \in \{0,1\}^\ell$ by invoking $\mathsf{Coin.Get}(\mathsf{ctx})$. Here r is unpredictable against the adversary when $t - f$ or fewer honest parties invoke the protocol. An (n,t)-threshold common coin protocol satisfies the following properties, except with negligible probability.

- *Termination.* If all honest parties invoke $\mathsf{Coin.Get}(\mathsf{ctx})$, then every honest party eventually outputs some value $r \in \{0,1\}^\kappa$.
- *Consistency.* For any fixed context ctx, if two honest parties output values r_i and r_j from $\mathsf{Coin.Get}(\mathsf{ctx})$, respectively, then $r_i = r_j$.
- *Pseudorandomness.* For any fixed context ctx, the adversary cannot predict the output r of $\mathsf{Coin.Get}(\mathsf{ctx})$ better than guessing over $\{0,1\}^\kappa$, if $t - f$ or fewer honest parties invoke the coin instance.

Instantiation. Throughout the paper, we adopt a (n,f)-threshold common coin protocol, instantiated by reconstructing the shared randomnesses that are preprocessed by the protocol described in Appendix D.5. Each context ctx selects a unique preprocessed value $[\![r_{\mathsf{ctx}}]\!]$, which can be collectively reconstructed by all parties using the robust reconstruction algorithm described in Appendix E, and subsequently mapped to the desired output domain (e.g., via a hash function to

$\{0,1\}^\kappa$). For notation simplicity, we treat ctx as implicit and write Coin.Get() for most cases.

Asynchronous verifiable information dispersal [19, 32, 73] is an asynchronous data-distribution primitive that allows a dealer to disperse a value $m \in \{0,1\}^*$ among parties such that the value can later be reconstructed, while ensuring that honest parties agree on whether the dispersal is valid. In an (n,t)-asynchronous verifiable information dispersal protocol, a dealer invokes Dispersal(m) to distribute the message m, and parties may later invoke Retrieve to recover the dispersed value. The protocol satisfies the following properties, except with negligible probability.

- *Termination.* If the dealer is honest and invokes Dispersal(m), then every honest party eventually completes the dispersal phase.
- *Agreement.* If one honest party completes the dispersal phase, then every honest party eventually completes the dispersal phase.
- *Availability.* If at least $t+1$ honest parties have completed the dispersal phase, then the dispersed value m is reconstructible from the information held by honest parties.
- *Correctness.* If the dealer is honest and disperses a value m, then any honest party invoking Retrieve() outputs m.
- *Consistency.* If two honest parties output values m_i and m_j from Retrieve(), respectively, then $m_i = m_j$.

Instantiation. We instantiate the AVID protocol from [19], which is based on erasure coding and a Merkle tree over the encoded fragments. In our protocol, we use Disperse$(\{m_{i,j}\}_{(i,j)\in I})$ to disperse an indexed sequence of messages, and parties may later invoke Retrieve(i,j) to retrieve the component associated with index i and j. Let M denote the canonical encoding of the entire indexed sequence. The dealer and the retrieving party each communicate $O(|M| + n\log n)$ bits, while each non-retrieving participant communicates $O(|M|/t + \log n)$ bits.

Asynchronous complete secret sharing [32, 69] is an asynchronous secret-sharing primitive that allows a dealer to share a secret $s \in \mathbb{Z}_p$ among parties such that the secret can later be reconstructed, while ensuring that honest parties agree on the validity and availability of the sharing. In an (n,t)-asynchronous complete secret sharing protocol, a dealer invokes Share(s) to distribute shares of the secret s, and parties may later invoke Reconstruct to recover the shared secret. The protocol satisfies the following properties, except with negligible probability.

- *Correctness.* If the dealer is honest, all honest parties will eventually output a share $\phi(i)$ where $\phi(\cdot)$ is a random polynomial with $\phi(0) = s$
- *Completeness.* If any honest party receives output, then there exists a unique degree-t polynomial $\tilde{\phi}(\cdot)$ such that each honest party eventually outputs $\tilde{\phi}(i)$. Moreover, if the dealer is honest, $\tilde{\phi}(0) = s$.
- *Secrecy.* If the dealer is honest, the adversary learns no information about the secret s beyond what is implied by its corrupted parties' shares.

Instantiation. We instantiate ACSS using the pairing-free construction in Algorithm 4, which combines generalized Pedersen commitments (Appendix D.3) with asynchronous verifiable information dispersal. This enables batched secret sharing with amortized $O(n)$ communication per secret, even against Byzantine adversaries.

C Concrete "Re-attempt Linkability" Attack under Asynchrony for ABs without Guaranteed Message Delivery

We observe that a naive extension of Blinder [4] to the asynchronous setting will introduce a non-negligible advantage for the adversary, enabling it to link messages to their original senders. Specifically, each client i in [4] independently selects a table position i and writes m_j and $(m_j)^2$ into position i of two secret-shared tables. Consequently, multiple clients may select the same position with non-negligible probability. Let M_i and $\hat{M}_i$ denote the reconstructed aggregates at position i in the two tables, respectively.

Then, servers first check whether $M_i^2 = \hat{M}_i$. If the equality holds, the entry is treated as containing a single message, and M_i is directly output. Otherwise, the entry is treated as containing two colliding messages $\tilde{m}_1$ and $\tilde{m}_2$, such that $M_i = \tilde{m}_1 + \tilde{m}_2$ and $\hat{M}_i = \tilde{m}_1^2 + \tilde{m}_2^2$. The servers then attempt to recover both messages by solving the resulting quadratic system. However, when more than two messages collide in the same entry, these two power sums no longer suffice to reconstruct the original messages. The recovery procedure then outputs two strings that do not correspond to any original message, which we refer to as a *delivery failure*.

The key concern is that a *delivery failure* may expose honest clients' messages to the adversary before these messages are successfully delivered. In an asynchronous setting, such a disclosure can be exploited to mount a linkability attack.[8] Specifically, the adversary first waits for delivery failures that reveal honest messages. It then selectively delays honest inputs to influence the set of clients participating in each epoch. If the same message is retransmitted in a later epoch, the adversary can identify it and narrow down its sender to the intersection of the candidate sender set from the earlier epoch and the adversarially influenced participant set of the later epoch.

[8] More generally, the same issue may arise in synchronous executions with a dynamic set of participating clients.

Table 3: Fraction of honest messages exposed before delivery under different ratios of corrupted clients, with input vector size $L = 2.75N$

Corruption ratio[†]	A1[‡]	A2[‡]	Leak[⋆]
1/3	0.55104%	2.07387%	2.62491%
1/2	1.24153%	2.52881%	3.77033%
2/3	2.21660%	2.30632%	4.52292%

[†] Corruption ratio F/N, where F and N denote the numbers of corrupted and participating clients, respectively.
[‡] Fractions of honest messages leaked through Cases A1 and A2, respectively.
[⋆] Total fraction of honest messages revealed to the adversary, computed as the sum of the leakage rates for Cases A1 and A2.

To make the attack concrete, we consider a collided message $m^\star$ first revealed in epoch e_0, with the adversary controlling F clients in every epoch. The attack (called "re-attempt linkability" attack by us) proceeds as follows.

- *Attack-Step 1: Early disclosure from failed collision recovery.* As discussed above, a *delivery failure* occurs when three or more messages are written to the same entry, causing the two-message recovery procedure to fail. However, because the adversary knows the messages contributed by corrupted clients, it can subtract their contributions from the reconstructed values M_i and $\hat{M}_i$ and recover the honest messages before their intended delivery in the following two cases:

  1. A1: one honest message $m^\star$ collides with multiple corrupted messages; or
  2. A2: two honest messages $m_1^\star, m_2^\star$ collide with one or more corrupted messages.

  In both cases, the adversary can subtract the known corrupted inputs from the decoded entry contents and obtain the message directly in advance.

  Moreover, such early leakage occurs with non-negligible probability. Table 3 reports the fraction of honest messages exposed before delivery under different corruption ratios.

- *Attack-Step 2: Adaptively filtering the candidate sender set.* Let $u^\star$ denote the honest sender of $m^\star$, and let H_0 be the set of honest clients whose inputs are included in epoch e_0. Since the adversary knows all messages submitted by corrupted clients but cannot distinguish among honest senders, the disclosure of $m^\star$ in epoch e_0 reveals only that $u^\star \in H_0$. The adversary then tests the clients in $H_0 = \{H_0[1], \ldots, H_0[|H_0|]\}$ one by one. In a subsequent epoch e_j, the adversary delays the inputs of all clients in $H_0 \setminus \{H_0[j]\}$, while allowing the input of $H_0[j]$ and sufficiently many inputs from clients outside H_0 to be processed.

- *Attack-Step 3: Identifying the sender across repeated epochs.* Since $m^\star$ remains undelivered after epoch e_0, its honest sender retransmits it in subsequent epochs. The adversary therefore checks whether the same message $m^\star$ appears in epoch e_j. If it does, then the only client from the original candidate set H_0 participating in that epoch is $H_0[j]$, and hence $u^\star = H_0[j]$.

As shown in Table 3, up to 4.52% of honest messages are exposed without being delivered and require clients to retransmit. Since approximately 95% of these retransmitted messages are successfully delivered in the subsequent epoch, the adversary can identify the reappearance and link them with controlled participant sets. Consequently, up to $4.52\% \times 95\% \approx 4.29\%$ of honest messages may become (eventually) linkable. In fact, the attack arises from the revelation of honest messages before delivery.

We note that our AAB definition, formalized by the ideal functionality $\mathcal{F}_{\mathsf{AAB}}$, fully addresses the above "re-attempt linkability" attack launched by an asynchronous adversary. Intuitively, $\mathcal{F}_{\mathsf{AAB}}$ ensures that all solicited messages are mixed and delivered, while all unsolicited messages remain private until they are included in a subsequent mixing instance. This prevents the adversary from gaining any advantage, as illustrated in the above attack through the client's repeated submissions of the same message.

D Offline Implementation DumboMPC++

This appendix describes DumboMPC++, our performance-optimized implementation for DumboMPC [67], as our offline-phase instantiation of AMPC, to support realistic deployment of mixing methods in an asynchronous network. Our construction eliminates the heavyweight pairing-based machinery required by KZG commitments and further improves the dominating verification cost of batched zero-knowledge proofs during triple generation by leveraging Bulletproofs. We begin with the underlying ACSS protocol, followed by AsyRanGen and AsyTriGen, which preprocess random secret sharings and random multiplication triples, respectively.

D.1 Recall DumboMPC

We consider two basic types of preprocessing tasks (i.e., shared randomness and shared multiplication triples) for the online phase and for preprocessing more involved correlated randomness. To our knowledge, DumboMPC is the state-of-the-art implementation for this purpose in the asynchronous G.O.D. setting considered by us.

DumboMPC primarily generates two types of correlated randomness: shared randomness and shared multiplication triples. As shown in Fig. 7, it adopts a classical BGW-style paradigm in the malicious setting, where it relies on

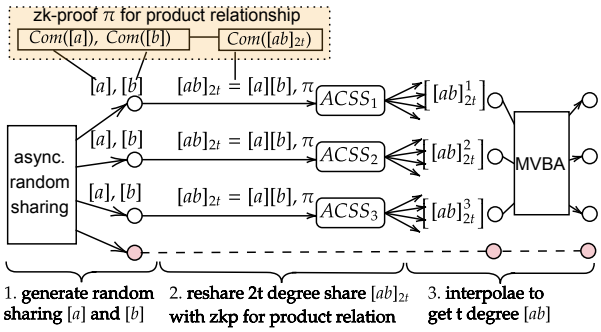


Figure 7: Asynchronous triple generation in DumboMPC

asynchronous complete secret sharing (ACSS) protocols [32, 69, 72] with homomorphic commitments together with zero-knowledge proofs for product relations to guarantee the correctness of the generated multiplication triples. In greater detail, it follows a three-step technical path: (i) build an Asynchronous Complete Secret Sharing (ACSS) scheme; (ii) use this ACSS to realize Asynchronous Shared Randomness Generation (AsyRanGen); and (iii) combine both to realize Asynchronous Triple Generation (AsyTriGen). The current instantiations cause two major performance bottlenecks:

- First, DumboMPC instantiates ACSS by hbACSS [69] with KZG commitments [50] for its compactness. When sharing a batch of $B = n^2$ secrets, the adoption of compact KZG commitments helps to achieve an amortized linear communication cost per secret. However, not only pairings themselves, but also other group operations over pairing-friendly curves, are substantially more expensive than their counterparts over standard elliptic curves without pairing support. This introduces considerable computational overhead and ultimately limits the preprocessing throughput of DumboMPC.
- Second, when in the batch preprocessing phase of multiplication triples (i.e., the AsyRanGen protocol), DumboMPC still requires that, for each multiplication triple, every party has to (independently) verify $O(n)$ zero-knowledge proofs from distinct parties to enforce correct product relations. When the protocol operates over a large batch to preprocess B triples in a single execution, each party must therefore verify $O(nB)$ such proofs, making verification another major computational bottleneck.

D.2 Overview of Improvements Introduced by DumboMPC++

Below, we briefly explain a few *optimizations* applied by us in DumboMPC++ to realize faster offline-phase preprocessing.

First, to mitigate the slow operations of pairing-friendly elliptic curves (required by KZG commitment), we start by replacing hbACSS with the GS23-ACSS protocol [45], which uses generalized Pedersen commitments (cf. §D.3) over standard elliptic curves with faster group operations. A remaining challenge is the quadratic per-share communication complexity in the worst case of GS23. Inspired by [69], we address this with a strategy of grouping a batch of B secrets into n groups, where each group is encrypted and dispersed independently. When the dealer is suspiciously malicious, a party only accuses that it receives a wrong share in one particular group, so other parties no longer have to reconstruct all groups to verify the accusation, saving the communication complexity of accusing a malicious dealer. We further use the batch reconstruction from hbACSS [69] so that recovering B secrets only requires each party to send $B/(t+1)$ field elements.

Second, in the AsyRanGen protocol, we replace the per-triple verification of the product relationship over committed values with a standard yet often overlooked technique of batched verification (cf. Appendix D.4). In particular, we leverage Bulletproofs-style zero-knowledge arguments for "arithmetic circuits" together with the Fiat–Shamir heuristic to prove the Hadamard product relation within a more compact proof. In our implementation, this reduces the per-product verification cost from 19 to about 15 group exponentiations, and shrinks the per-product proof size from 6 group elements and 8 field elements to approximately 4 group elements and 7 field elements.

D.3 First Ingredient: Batch (Generalized) Pedersen Polynomial Commitment

We now recall the generalized Pedersen commitment scheme, which serves as the building block for the batch polynomial commitment (PBC) scheme [45, 61], later to be used in our ACSS instantiation in Appendix D.5. Let $\mathbb{G}$ be a cyclic group of prime order p, and let $\{g_1, \ldots, g_B, h\} \subset \mathbb{G}$ be a set of independently sampled generators. For a message vector $\mathbf{a} = (a_1, \ldots, a_B) \in \mathbb{Z}_p^B$ and an opening value $\hat{a} \in \mathbb{Z}_p$, the generalized Pedersen commitment is defined as:

$$\mathsf{Com}(\mathbf{a}, \hat{a}) := h^{\hat{a}} \cdot \prod_{j=1}^{B} g_j^{a_j}.$$

For $B = 1$, this reduces to the standard Pedersen commitment.

Batch polynomial commitment (BPC) from generalized Pedersen commitment. Let $\{\phi_j(X)\}_{j\in[B]}$ be B polynomials over $\mathbb{Z}_p$ of degree at most t, where $\phi_j(X) = \sum_{k=0}^{t} a_{j,k} X^k$. Then, the prover samples a random hiding polynomial $\hat{\phi}(X) = \sum_{k=0}^{t} \hat{a}_k X^k$ of degree at most t. The scheme consists of the following algorithms.

- $\mathsf{BatchCommit}(\{\phi_j\}_{j\in[B]}, \hat{\phi}) \to \mathbf{C}$. Given the message polynomials and the hiding polynomial, the algorithm outputs a $(t+1)$-size vector of Pedersen commitments $\mathbf{C} = (C_0, \ldots, C_t) \in \mathbb{G}^{t+1}$, where each C_k commits to the k-th

coefficients of all B polynomials:

$$C_k = \mathsf{Com}\big((a_{1,k},\ldots,a_{B,k}),\hat{a}_k\big) = h^{\hat{a}_k}\cdot\prod_{j=1}^{B} g_j^{a_{j,k}}.$$

- $\mathsf{BatchOpen}(\{\phi_j\}_{j\in[B]},\hat{\phi},i)\to(\{y_j\}_{j\in[B]},\hat{y})$. Given an evaluation point $i\in\mathbb{Z}_p$, the opening algorithm outputs

$$y_j=\phi_j(i)\quad\text{for all } j\in[B],\qquad \hat{y}=\hat{\phi}(i).$$

- $\mathsf{BatchVerify}(\mathbf{C},i,\{y_j\}_{j\in[B]},\hat{y})\to\{0,1\}$. Given the commitment vector $\mathbf{C}$, an evaluation point $i\in\mathbb{Z}_p$, claimed evaluations $\{y_j\}_{j\in[B]}$, and a hiding evaluation $\hat{y}$, the verifier accepts if and only if $\prod_{k=0}^{t}(C_k)^{i^k}=h^{\hat{y}}\cdot\prod_{j=1}^{B}g_j^{y_j}$.

Properties. The above batch polynomial commitment satisfies *correctness*, *perfect hiding*, and *computational binding*.

- *Correctness.* For any B polynomials $\{\phi_j\}_{j\in[B]}$, any hiding polynomial $\hat{\phi}$, and any evaluation point $i\in\mathbb{Z}_p$, if we have $\mathbf{C}\leftarrow\mathsf{BatchCommit}(\{\phi_j\}_{j\in[B]},\hat{\phi})$ and $(\{y_j\}_{j\in[B]},\hat{y})\leftarrow\mathsf{BatchOpen}(\{\phi_j\}_{j\in[B]},\hat{\phi},i)$, then $\mathsf{BatchVerify}(\mathbf{C},i,\{y_j\}_{j\in[B]},\hat{y})=1$.
- *Perfect hiding.* The commitment vector $\mathbf{C}$ reveals no information about the committed polynomials $\{\phi_j\}_{j\in[B]}$. More precisely, for any two batches of polynomials $\{\phi_j\}_{j\in[B]}$ and $\{\phi'_j\}_{j\in[B]}$ of degree at most t, the distributions of $\mathsf{BatchCommit}(\{\phi_j\}_{j\in[B]},\hat{\phi})$ and $\mathsf{BatchCommit}(\{\phi'_j\}_{j\in[B]},\hat{\phi}')$ are identical when $\hat{\phi}$ and $\hat{\phi}'$ are sampled uniformly at random.
- *Computational binding.* Under the discrete-logarithm assumption, no probabilistic polynomial-time adversary can output a commitment vector $\mathbf{C}$, an evaluation point i, and two distinct valid openings $(\{y_j\}_{j\in[B]},\hat{y})\neq(\{y'_j\}_{j\in[B]},\hat{y}')$ such that $\mathsf{BatchVerify}(\mathbf{C},i,\{y_j\}_{j\in[B]},\hat{y})=1$ and $\mathsf{BatchVerify}(\mathbf{C},i,\{y'_j\}_{j\in[B]},\hat{y}')=1$, except with negligible probability.

In addition, the polynomial commitment scheme described above is also additively homomorphic. That said, given two commitments $\mathbf{C}=(C_0,\ldots,C_t)$ and $\mathbf{C}'=(C'_0,\ldots,C'_t)$ for polynomials $\{\phi\}_{j\in[B]}$ and polynomials $\{\psi_j\}_{j\in[B]}$, respectively, $((C_0)^{\alpha}(C'_0)^{\beta},\ldots,(C_t)^{\alpha}(C'_t)^{\beta})$ is a polynomial commitment for the batch of polynomials $\{\alpha\phi_j+\beta\psi_j\}_{j\in[B]}$.

D.4 Second Ingredient: Batched Proof of Product Relationship over BPC

Now we detail another implementation-level performance optimization for efficiently verifying a batch of product relation proofs that will be used in our preprocessing protocol for multiplication triples.

Proof of Product Relationship. A major computational task in the DumboMPC multiplication triple generation protocol is generating and verifying proofs with respect to the following relation, where g and h are independent generators of a cyclic group $\mathbb{G}$ of prime order q:

$$\left\{\begin{array}{l}\text{public: } \{(\mathsf{Ped}_{a_k},\mathsf{Ped}_{b_k},\mathsf{Ped}_{c_k})\in\mathbb{G}^3\}_{k\in[B]},g,h\\ \text{witness: } \{(a_k,\tilde{a}_k,b_k,\tilde{b}_k,c_k,\tilde{c}_k)\in\mathbb{Z}_p^6\}_{k\in[B]}\\ \text{relation: } \forall k\in[B],\ c_k=a_k\cdot b_k\ \wedge\ \mathsf{Ped}_{a_k}=g^{a_k}h^{\tilde{a}_k}\\ \qquad\qquad\wedge\ \mathsf{Ped}_{b_k}=g^{b_k}h^{\tilde{b}_k}\ \wedge\ \mathsf{Ped}_{c_k}=g^{c_k}h^{\tilde{c}_k}.\end{array}\right\}$$

In DumboMPC, this relation is proven naively in a tuple-by-tuple manner. Specifically, for each $k\in[B]$, an individual proof π_k is generated and verified for the statement $(\mathsf{Ped}_{a_k},\mathsf{Ped}_{b_k},\mathsf{Ped}_{c_k})$.

Batched Proof of Product Relationship from Bulletproof. To reduce both proof size and verification complexity, we batch the B multiplicative relations into a single argument based on the arithmetic-circuit variant of Bulletproofs (cf. Section 5 of [16]). Specifically, it provides a proof system for arbitrary circuits where input wires can come from Pedersen commitments as follows

$$\left\{\begin{array}{l}\text{public: } g,h\in\mathbb{G},\ \boldsymbol{g},\boldsymbol{h}\in\mathbb{G}^B,\ \mathbf{Ped}\in\mathbb{G}^V,\\ \qquad\mathbf{W}_a,\mathbf{W}_b,\mathbf{W}_c\in\mathbb{Z}_p^{Q\times B},\ \mathbf{W}\in\mathbb{Z}_p^{Q\times V},\ \boldsymbol{\chi}\in\mathbb{Z}_p^{Q};\\ \text{witness: } \boldsymbol{a},\boldsymbol{b},\boldsymbol{c}\in\mathbb{Z}_p^B,\ \boldsymbol{v},\boldsymbol{\gamma}\in\mathbb{Z}_p^V;\\ \text{relation: } \mathsf{Ped}_i=g^{v_i}h^{\gamma_i}\quad\forall i\in[V]\ \wedge\\ \qquad\boldsymbol{a}\circ\boldsymbol{b}=\boldsymbol{c}\ \wedge\ \mathbf{W}_a\boldsymbol{a}+\mathbf{W}_b\boldsymbol{b}+\mathbf{W}_c\boldsymbol{c}=\mathbf{W}\boldsymbol{v}+\boldsymbol{\chi}.\end{array}\right\}$$

Here, B denotes the number of multiplication gates, V denotes the number of externally committed values, and Q denotes the number of linear consistency constraints. The matrices $\mathbf{W}_a$, $\mathbf{W}_b$, and $\mathbf{W}_c$ specify the coefficients of the left-input, right-input, and output wires, respectively. The matrix $\mathbf{W}$ specifies the coefficients of the values committed in Ped, while $\boldsymbol{\chi}$ is a public constant vector.

For our committed Hadamard product relation, each tuple contributes three externally committed values. We therefore set $V=3B$ and $Q=3B$, arrange their openings as $\boldsymbol{v}=(\boldsymbol{a}\,\|\,\boldsymbol{b}\,\|\,\boldsymbol{c})$. Let I_B denote the B-dimensional identity matrix, and let 0_B denote the all-zero matrix. We instantiate the consistency matrices as

$$\mathbf{W}_a=\begin{pmatrix}I_B\\0_B\\0_B\end{pmatrix},\qquad \mathbf{W}_b=\begin{pmatrix}0_B\\I_B\\0_B\end{pmatrix},\qquad \mathbf{W}_c=\begin{pmatrix}0_B\\0_B\\I_B\end{pmatrix},$$

and $\mathbf{W}=I_{3B}$ and $\boldsymbol{\chi}=0$. Under this initialization, it binds the multiplication-gate wires to their corresponding standard Pedersen commitments. Consequently, the remaining circuit constraint is precisely the committed Hadamard-product relation $\boldsymbol{a}\circ\boldsymbol{b}=\boldsymbol{c}$, where $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are exactly committed to

$\{\mathsf{Ped}_i\}_{i\in[1,B]}$, $\{\mathsf{Ped}_i\}_{i\in[B+1,2B]}$, and $\{\mathsf{Ped}_i\}_{i\in[2B+1,3B]}$, respectively.

In the triple generation protocol, we must still prove consistency between the vector of commitments (appearing the Bulletproof statement) and BPCs (derived from underlying GS23-ACSS protocols), such that the values committed to $\{\mathsf{Ped}_i\}_{i\in[1,B]}$, $\{\mathsf{Ped}_i\}_{i\in[B+1,2B]}$, and $\{\mathsf{Ped}_i\}_{i\in[2B+1,3B]}$ are same with the values committed to three BPCs for $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$, respectively. This can be achieved using a standard Σ-protocol combined with the Fiat-Shamir transformation. Although the asymptotic proof size of our batched proof remains $O(B)$, the concrete efficiency gains are significant: the amortized verification cost per product is reduced from 19 to approximately 15 group exponentiations, and the amortized proof size per product drops from 6 group elements and 8 field elements to approximately 4 group elements and 7 field elements. The full description and security analysis of the triple generation protocol will soon be discussed in Appendix D.7.

D.5 Step 1: Performance-Improved ACSS Instantiation from Batch BPC

We now present how DumboMPC++ instantiates the ACSS protocol by improving a construction from Groth and Shoup (GS23) [45] that uses generalized Pedersen's polynomial commitment (described in Appendix D.3), through a group-based strategy combined with batched reconstruction (from hbACSS [69]) to obtain linear amortized communication complexity even against active malicious adversaries. Note that DumboMPC adopts a KZG-based ACSS instantiation instead of GS23, as the original GS23 protocol has an amortized quadratic communication complexity in the worst-case when an active adversary launches a performance downgrade attack, which will be addressed in this appendix.

INGREDIENTS. Our improved instantiation of GS23 ACSS is built from the following standard components:

- The generalized Pedersen commitment described in the previous Appendix D.3
- An RBC protocol [19] and an IND-CPA-secure public-key encryption scheme, instantiated via hybrid encryption to support arbitrary-length messages, as in [69].
- An asynchronous verifiable information dispersal (AVID) protocol [19], consisting of the subroutines Disperse and Retrieve.

THE ACSS PROTOCOL. We now describe the improved ACSS construction; the detailed procedure is given in Algorithm 4

- *Sharing phase*. The dealer P_ℓ samples a batch of B secrets and encodes each secret as the zero-point value at a random degree-t polynomial. It then partitions the resulting polynomials into $t+1$ disjoint groups. For each group, the dealer computes a batch commitment using generalized Pedersen commitments and disseminates the commitments via reliable broadcast. Finally, the dealer encrypts the corresponding shares for each receiver and disperses the resulting ciphertexts using AVID, ensuring asynchronous and verifiable delivery.
- *Share validation*. Each receiver P_i retrieves its encrypted payloads, decrypts them using its secret key $sk_{i\ell}$, and checks that the recovered shares are consistent with the corresponding commitments. If any check fails, P_i issues a public accusation by revealing $sk_{i\ell}$.
- *Handling Implication*. Upon receiving an accusation, honest parties verify its validity by: (1) check that the correctness of the revealed secret key $sk_{i\ell}$ matches the public key $pk_{i\ell}$; (2) retrieve the ciphertext for the corresponding group; (3) decrypt the ciphertext; and (4) check the recovered share against the dealer's commitment. If the accusation is valid, all honest parties invoke the share-recovery subroutine.
- *Share recovery*. If the dealer's sharing is found invalid (by some honest party), the honest party would reliably broadcast an Implicate message to ask all honest parties to collaborate to reconstruct the missing shares. Concretely, assuming B is divisible by $t+1$, they arrange the B univariate sharing polynomials into $d = B/(t+1)$ degree-(t,t) bivariate polynomials, enabling the missing shares to be recovered through batched interpolation. Once reconstruction completes, each receiver outputs a consistent batch of corrected shares that matches the original commitments.

Algorithm 4 Worst-Case-Improved GS23 ACSS Scheme

Setup:

1: Each party P_i chooses n PKE key pairs $\{(sk_{ij}, pk_{ij})\}_{j\in[n]}$ and publishes these n public keys $\{pk_{ij}\}_{j\in[n]}$, where $pk_{i\ell}$ is used by the dealer P_ℓ to encrypt shares sent to P_i.
2: Public parameter of generalized Pedersen Commitment $\{g_0,\ldots,g_B,h\}$.
3: Publish the public keys $\{pk_{ij}\}_{i,j\in[n]}$ to all parties.

Procedure for dealer P_ℓ

4: Upon P_ℓ revives a list of input secrets $(s_1,\ldots,s_B)$, where each $s_k \in \mathbb{Z}_p$, it executes as follows
5: Let $d \leftarrow B/(t+1)$ ▷ Evenly divide B input secrets into $t+1$ groups. Note: pad the input by zeros such that $(t+1) \mid B$, if needed.
6: Set $G_j \leftarrow \{(j-1)d+1,\ldots,jd\}$, for $j \in [t+1]$ ▷ G_j is a d-size set of indices representing the j-th group of secrets
7: For each $j \in [t+1]$, P_ℓ sample a random degree-t polynomial $\hat{\phi}_j(X) \in \mathbb{Z}_p[X]$ such that $\hat{\phi}_k(0) = \hat{s}_k$ ▷ Here $\hat{\phi}_j(X)$ is the hiding polynomial for the j-th group of secrets
8: For each $k \in [B]$, sample a random degree-t polynomial $\phi_k(X) \in \mathbb{Z}_p[X]$ such that $\phi_k(0) = s_k$ ▷ Here $\phi_k(X)$ encodes the k-th input secret as its zero-point evaluation
9: $\mathbb{C} \leftarrow \left\{\mathbf{C}_j := \mathsf{PC.BatchCommit}(\{\phi_k(\cdot)\}_{k\in G_j}, \hat{\phi}_j(\cdot))\right\}_{j\in[t+1]}$

RBC($\mathbb{C}$)
for $i = 1$ to n
  for $j = 1$ to $(t+1)$
    $z_{i,j} \leftarrow \mathsf{Enc}_{pk_{i\ell}}(\{\phi_k(i)\}_{k\in G_j}, \hat{\phi}_j(i))$
Disperse$(\{z_{i,j}\}_{i\in[n],j\in[t+1]})$

Procedure for every receiver P_i

// Wait for broadcasts
Wait to receive $\mathbb{C} = \{\mathbf{C}_1, \cdots, \mathbf{C}_{t+1}\}$ from RBC
Wait for Disperse to complete
// Decrypt and validate
for $j = 1$ to $t+1$
  $z_{i,j} \leftarrow \mathsf{Retrieve}(i,j)$
  $(\{\phi_k(i)\}_{k\in G_j}, \hat{\phi}_j(i)) \leftarrow \mathsf{Dec}_{sk_{i\ell}}(z_{i,j})$
  if any PC.BatchVerify$(\mathbf{C}_j, i, \{\phi_k(i)\}_{k\in G_j}, \hat{\phi}_j(i)) \neq 1$ or decryption fails
    RBC(Implicate, $sk_{i\ell}, j$) ▷ If $sk_{i\ell}$ is expected to be reused throughout many ACSS instances, P_i can compute a zero-knowledge proof for correct decryption instead of directly sending the PKE's private decryption key $sk_{i\ell}$, only revealing the one-time symmetric key used in the "hybrid" PKE.
Upon receiving all valid shares $\{\mathbf{C}_j, \{\phi_k(i)\}_{k\in G_j}, \hat{\phi}_j(i)\}_{j\in[d]}$
  sendall OK
// Bracha-style agreement
On receiving OK from $2t+1$ parties,
  sendall READY
If P_i receives READY from $t+1$ parties and has not sent READY:
  sendall READY
Wait to receive READY from $2t+1$ parties,
  if all shares are valid **then**
    output $sh = \{\mathbf{C}_j, \{\phi_k(i)\}_{k\in G_j}, \hat{\phi}_j(i)\}_{j\in[t+1]}$

Procedure for receiver P_i with output to handle implication

// Handling Implication
On receiving (Implicate, $sk_{j\ell}, k$) from P_j,
  Ignore if already in Share Recovery or $pk_{j\ell} \neq g^{sk_{j\ell}}$
  $z_{j,k} \leftarrow \mathsf{Retrieve}(j,k)$ ▷ Each party assists with Retrieve(j,k) only after receiving RBC(Implicate, $sk_{j\ell}, k$) from P_j.
  if PC.BatchVerify$(\mathbf{C}_k, j, \{\phi_u(j)\}_{u\in G_k}, \hat{\phi}_k(j)) \neq 1$ or decryption fails
    Proceed to Share Recovery

Procedure for every receiver P_i to share recovery

// Share Recovery ▷ Recover all shares for P_j
Interpolate $\{\mathbf{C}'_k\}_{k\in[n]}$ from $\{\mathbf{C}_k\}_{k\in[t+1]}$
if P_i have received valid share **then**
  for $P_j \in \mathcal{P}$
    Intepolate $\phi_k(i,\cdot)$ from $\{u, \phi_k(i)\}_{u\in[t+1]}$
    Intepolate $\hat{\phi}(i,\cdot)$ from $\{u, \hat{\phi}(i)\}_{u\in[t+1]}$
    send $(R_1, \{\phi_k(i,j)\}_{k\in[d]}, \hat{\phi}(i,j))$ to P_j
// R_1 phase
Set $\mathcal{R}_{1i} \leftarrow \emptyset$
Upon receiving $(R_1, \{\phi_k(j,i)\}_{k\in[d]}, \hat{\phi}(j,i))$ from P_j
  if PC.BatchVerify$(\mathbf{C}'_k, i, \{\phi_k(j,i)\}_{k\in[d]}, \hat{\phi}(j,i)) = 1$
    $\mathcal{R}_{1i} \leftarrow \mathcal{R}_{1i} \bigcup \{j\}$
    if $|\mathcal{R}_{1i}| \geq t+1$
      Interpolate $\{\phi_k(\cdot,i)\}_{k\in[d]}$ and $\hat{\phi}(\cdot,i)$
      for each $P_j \in \mathcal{P}$
        send $(R_2, \{\phi_k(j,i)\}_{k\in[d]}, \hat{\phi}(\cdot,i))$ to P_j
// R_2 phase
Set $\mathcal{R}_{2i} \leftarrow \emptyset$
Upon receiving $(R_2, \{\phi_k(j,i)\}_{k\in[d]}, \hat{\phi}(\cdot,i))$ from P_j
  $\mathcal{R}_{2i} \leftarrow \mathcal{R}_{2i} \bigcup \{j\}$
  if $|\mathcal{R}_{2i}| \geq 2t+1$
    Roubustly interpolate $\{\phi_k(i,\cdot)\}_{k\in[d]}$ and $\hat{\phi}(i,\cdot)$
    output $sh = \{\mathbf{C}_j, \{\phi_k(i,j)\}_{k\in G_j}, \hat{\phi}(i,j)\}_{j\in[t+1]}$

Communication complexity of ACSS**.** We instantiate ACSS using RBC and AVID. RBC is used to consistently deliver the commitment vector, incurring amortized communication $O(n|M| + \kappa n^2 \log n)$, while AVID disperses encrypted shares with communication $O(|M| + n\log n)$ for the sender and each receiver, and $O(|M|/t + \log n)$ for each remaining participant.

For each ACSS invocation, the dealer needs to broadcast $t+1$ groups of polynomial commitments via RBC. The total commitment size is $O(\kappa n^2)$, and hence the RBC communication is $O(\kappa n^3 + \kappa n^2 \log n)$. The dealer also disperses encrypted shares of total size $O(\kappa nB)$ through AVID. Thus, in the good case, the total communication per dealer is $O(\kappa nB + \kappa n^3 + \kappa n^2 \log n)$, which gives $O(n)$ elements per secret, when $B = \Omega(n^2)$.

In the presence of Byzantine behavior, the adversary may increase communication by causing honest parties to process implications and, in the worst case, to enter the share-recovery phase. A malicious dealer may distribute invalid encrypted shares, and corrupted receivers may issue false implications to force honest parties to invoke Retrieve for validation. However, we require honest parties only process implications delivered in the corresponding RBC instance, assist a retrieval only when the revealed secret key matches the corresponding published public key, and process at most one implication for each receiver-dealer pair. Additional implications for other groups from the same receiver against the same dealer are ignored, since one invalid group already suffices to trigger the recovery decision.

Specifically, for a fixed dealer, each party invokes Retrieve to handle processed implications at most $O(n)$ times, incurring $O(nd + n^2 \log n)$ communication as the retrieving party. The same party incurs $O(n^2(d/t + \log n))$ communication as an assisting participant. Since $t = \Omega(n)$ and $d = B/(t+1)$, the total implication-handling cost per party is $O(B + n^2 \log n)$. When $B = \Omega(n^2 \log n)$, this contributes only $O(n)$ amortized communication per secret. If share recovery is triggered, batch reconstruction requires each party to send only $O(d)$ additional payload, which is dominated by the same bound. Therefore, implication handling and share recovery do not change the $O(n)$ amortized communication per secret, even in the worst case.

Security of improved ACSS. The security properties of our improved ACSS can be summarized as the next theorem.

Theorem 1 (Correctness, Completeness, Secrecy)**.** *If n parties participate in the* ACSS *protocol of Algorithm 4 with dealer P_ℓ on input $(s_1,\ldots,s_B) \in \mathbb{Z}_p^B$, and the adversary corrupts at most $t < n/3$ parties, then, except with negligible probability in κ, the following properties hold.*

- ***Correctness***. *If the dealer P_ℓ is honest, then every honest party P_i eventually outputs shares $\{\phi_j(i)\}_{j\in[B]}$ such that, for every $j \in [B]$, ϕ_j is a uniformly random degree-t polynomial over $\mathbb{Z}_p$ satisfying $\phi_j(0) = s_j$.*
- ***Completeness***. *If any honest party terminates the Bracha-style agreement phase of* ACSS *and outputs a batch of shares $\{\phi_j(i)\}_{j\in[B]}$, then there exists a unique batch of degree-t polynomials $\{\phi'_j\}_{j\in[B]}$ such that every honest party eventually terminates and outputs $\{\phi'_j(i)\}_{j\in[B]}$.*
- ***Secrecy***. *If the dealer P_ℓ is honest, then any probabilistic polynomial-time adversary cannot learn any information about any shared secret in $\{s_j\}_{j\in[B]}$.*

Proof. We now prove the correctness, completeness, and secrecy in turn.

Correctness. The correctness property follows directly. If the dealer P_ℓ is honest, then both RBC and AVID complete successfully, and every honest party receives valid shares and outputs the corresponding valid shares.

Completeness. We now prove that the ACSS protocol satisfies the property of *Completeness*, if any honest party P_i terminates and outputs a batch of shares, then every honest party P_k eventually terminates and outputs a consistent set of shares $\{\phi_j(k)\}_{j\in[B]}$.

We first show that if any honest party outputs, then all honest parties eventually complete the Bracha-style agreement. Suppose that an honest party P_i outputs a batch of shares. By the protocol, P_i must have received READY messages from at least $2t+1$ distinct parties, at least $t+1$ of whom are honest. Since every honest sender broadcasts READY to all parties, every honest party eventually receives these $t+1$ READY messages. Any honest party that has not yet broadcast READY then does so upon receiving READY messages from $t+1$ distinct parties. Consequently, all honest parties eventually broadcast READY. Since there are at least $n-t \geq 2t+1$ honest parties, every honest party eventually receives at least $2t+1$ READY messages and completes the agreement phase.

It remains to show that every honest party eventually outputs and that all such outputs are consistent with the same unique batch of degree-t polynomials. We distinguish two cases according to whether the dealer is honest or Byzantine.

- *Honest dealer.* If the dealer is honest, then it samples a well-defined batch of degree-t polynomials $\{\phi_j\}_{j\in[B]}$, distributes the corresponding shares, and commits to them correctly in $\mathbb{C}$. By the correctness and termination properties of the AVID and RBC primitives, every honest party eventually receives its encrypted share together with the corresponding commitment $\mathbb{C}$. After decryption, every honest party obtains its batch of shares $\{\phi_j(i)\}_{j\in[B]}$, which verifies against $\mathbb{C}$. Therefore, all honest parties eventually output shares that are consistent with the same batch of degree-t polynomials.
- *Faulty dealer.* Suppose that the dealer is Byzantine. Since some honest party P_i terminates and outputs, P_i must have received READY messages from at least $2t+1$ distinct parties. Among these parties, at least $t+1$ are honest. Consider the first honest party that broadcasts READY. Since at most t Byzantine parties could have broadcast READY beforehand, its READY message cannot have been triggered by the $t+1$-READY relay rule. Hence, it must have been triggered by the initial READY condition, namely, after receiving OK messages from at least $n-t$ distinct parties. Among these parties, at least $n-2t \geq t+1$ are honest. Since an honest party broadcasts OK only after obtaining and validating its own batch of shares against the commitment $\mathbb{C}$, there exist at least $t+1$ honest parties holding shares that verify against the same commitment $\mathbb{C}$. For each $j \in [B]$, the valid shares held by these $t+1$ honest parties uniquely determine a degree-t polynomial, denoted by ϕ'_j. Thus, these shares define a unique batch of degree-t polynomials $\{\phi'_j\}_{j\in[B]}$. By the binding property of the commitment $\mathbb{C}$, any share that verifies against $\mathbb{C}$ must be consistent with this same batch of polynomials.

  Now consider any party that broadcasts an implication message. The implication message specifies the index of the challenged group and reveals the corresponding decryption key. By the validity and termination properties of RBC, all honest parties eventually receive the same implication message. They can then invoke Retrieve to consistently obtain the ciphertext corresponding to the implication. Next, those honest parties verify whether the revealed key is consistent with the published public key, decrypt it using the revealed key, and verify whether the resulting share is consistent with the published commitment. If the implication is invalid, it is ignored. If it is valid, all honest parties will move to the share recovery phase. Since at least $t+1$ honest parties hold valid shares consistent with $\mathbb{C}$, the recovery procedure can reconstruct, for every $j \in [B]$, the unique degree-t polynomial ϕ'_j, and therefore enables each honest party P_i to recover its own evaluation $\{\phi'_j(i)\}_{j\in[B]}$. Consequently, every honest party eventually outputs shares consistent with the same unique batch of degree-t polynomials.

Combining the arguments above, if any honest party terminates and outputs a batch of shares, then there exists a unique batch of degree-t polynomials $\{\phi'_j\}_{j\in[B]}$ such that every honest party eventually terminates and outputs $\{\phi'_j(i)\}_{j\in[B]}$.

Secrecy. We now prove that, when the dealer P_ℓ is honest, then any probabilistic polynomial-time adversary cannot learn any information about any shared secret in $\{s_j\}_{j\in[B]}$.

Let $\mathcal{P}_\mathcal{C}$ denote the set of corrupted parties, where $|\mathcal{P}_\mathcal{C}| \le t$. We argue by a sequence of Games.

Game 0. This is the real execution of the ACSS protocol with an honest dealer.

Game 1. The simulator $\mathcal{S}$ samples $\alpha_1,\ldots,\alpha_B,\tau \xleftarrow{\$} \mathbb{Z}_p^*$, sets $g_k := g^{\alpha_k}$ for $k \in [B]$ and $h := g^\tau$, and publishes the public parameters $\mathsf{pp} = (g_1,\ldots,g_B,h)$. It then runs the rest of the ACSS protocol exactly as in Game 0. Since $g_1,\ldots,g_B,h$ are uniformly random generators of $\mathbb{G}$, and the distribution is identical to the real setup, Game 0 and Game 1 are identically distributed.

Game 2. In this game, $\mathcal{S}$ replaces the honest dealer's sharing polynomials with simulated polynomials whose constant terms are independent of the real secrets while preserving all shares held by corrupted parties.

For each $j \in [B]$, the simulator first samples $\psi_j(0) \xleftarrow{\$} \mathbb{Z}_p$ independently and uniformly at random, and then samples a degree-t polynomial $\psi_j(X)$ subject to $\psi_j(i) = \phi_j(i)$, for every $P_i \in \mathcal{P}_\mathcal{C}$. Such a polynomial exists since $|\mathcal{P}_\mathcal{C}| \le t$, so the random constant term together with the evaluations of the corrupted parties imposes at most $t+1$ interpolation constraints.

To preserve the generalized Pedersen commitments, $\mathcal{S}$ uses the trapdoor $(\alpha_1,\ldots,\alpha_B,\tau)$ to define

$$\hat{\psi}(X) = \hat{\phi}(X) + \tau^{-1}\sum_{j=1}^{B}\alpha_j\big(\phi_j(X) - \psi_j(X)\big).$$

Thus, all generalized Pedersen commitments remain unchanged.

Moreover, for every corrupted party $P_i \in \mathcal{P}_\mathcal{C}$, we have $\psi_j(i) = \phi_j(i)$ for all $j \in [B]$. Substituting these equalities into (4) gives $\hat{\psi}(i) = \hat{\phi}(i)$. Hence, the complete shares held by corrupted parties are identical to those in Game 1.

The simulator then forms the ciphertexts and the corresponding dispersed data using the simulated shares. Ciphertexts addressed to corrupted parties encrypt exactly the same shares as in Game 1, whereas ciphertexts addressed to honest parties encrypt the corresponding simulated shares. All protocol values derived from these ciphertexts are generated consistently with the simulated execution.

By the IND-CPA security of the encryption scheme, ciphertexts encrypting the simulated shares of honest parties are computationally indistinguishable from ciphertexts encrypting their original shares. Equivalently, this follows by a standard hybrid argument over all ciphertexts addressed to honest parties. Therefore, the adversary's view in Game 2 is computationally indistinguishable from its view in Game 1.

Finally, in Game 2, the values $\{\psi_j(0)\}_{j\in[B]}$ are independent uniformly random values and hence independent of the real secrets $\{s_j\}_{j\in[B]}$, while the shares of corrupted parties remain unchanged. The public commitments are also unchanged, and the ciphertexts corresponding to honest parties hide their underlying shares by IND-CPA security. Consequently, the adversary's view in Game 2 is independent of the real secrets. □

D.6 Step 2: Asynchronous Shared Randomness Generation using New ACSS

AsyRanGen Protocol. The AsyRanGen protocol leverages our improved ACSS as a core building block to generate shared randomness. The protocol proceeds in three phases:

1. Each party $P_i \in \mathcal{P}$ acts as a dealer to instantiate an ACSS instance, sharing a batch of B secrets. Simultaneously, each P_i participates as a receiver in the instances initiated by all other parties.
2. The parties execute an MVBA protocol to reach consensus on a common subset $\mathcal{T} \subset \mathcal{P}$ of at least $n-t$ successfully completed ACSS instances.
3. Upon obtaining the output shares from the agreed subset $\mathcal{T}$, the parties apply a fixed Vandermonde-based linear transformation to the shares, the blinding shares, and the corresponding commitments. This extraction process lets the parties derive a batch of $B\cdot(t+1)$ secretly-shared uniformly random values, together with a generalized Pedersen polynomial commitment that commits to all of them in a batched manner.

More concretely, the extraction is applied not only to the shared values but also to the associated hiding values and commitments. Since $|\mathcal{T}| \ge n-t \ge 2t+1$, let $\mathcal{J} = (\mathcal{J}_1,\ldots,\mathcal{J}_{2t+1})$ be the canonical ordered subset of $2t+1$ completed ACSS instances selected from $\mathcal{T}$. Let $\mathbf{V} \in \mathbb{Z}_p^{(t+1)\times(2t+1)}$ be the fixed Vandermonde matrix. For each output row $\rho \in [t+1]$ and each batch index $k \in [B]$, the parties define the extracted sharing polynomial as

$$\phi_{\rho,k} = \sum_{j=1}^{2t+1} V_{\rho,j}\phi_k^{(\mathcal{J}_j)}.$$

Then the corresponding hiding polynomial is defined using the same coefficients

$$\hat{\phi}_{\rho,k} = \sum_{j=1}^{2t+1} V_{\rho,j}\hat{\phi}_k^{(\mathcal{J}_j)}.$$

Moreover, the public commitments to these extracted sharing polynomials are computed homomorphically with the same coefficients, namely as the component-wise product of the corresponding input commitments raised to $V_{\rho,j}$. Since generalized Pedersen commitments are additively homomorphic, these combined commitments are exactly commitments to the extracted sharing polynomials together with the extracted hiding polynomials.

This batch extraction leverages the algebraic structure of the underlying secret sharing scheme to produce $B(t+1) =$

$O(nB)$ shared random values. As shown in the communication analysis below, this reduces the amortized communication complexity per shared randomness to $O(n)$.

The detailed communication complexity analysis and formal security proofs are given below.

Communication complexity of AsyRanGen**.** The communication cost of AsyRanGen is dominated by n parallel ACSS instances and a single MVBA invocation. As established in § D.5, each ACSS instance incurs $O(nB)$ total communication for a batch of B secrets. Thus, the n parallel ACSS instances incur $O(n^2B)$ total communication. The MVBA invocation incurs $O(\kappa n^3)$ bits. The extraction phase is local and introduces no additional communication. Since the protocol outputs $(t+1)B$ shared random values, the amortized total communication per shared random value is $O(n)+O\left(\frac{\kappa n^2}{B}\right)$. For a sufficiently large batch size B that amortizes the MVBA overhead, this yields $O(n)$ amortized communication per shared random value.

Security of AsyRanGen. The security properties of AsyRanGen are summarized by the following theorem.

Theorem 2 (Termination, Consistency, Secrecy and Randomness)**.** *Let n parties execute the* AsyRanGen *protocol in an asynchronous network in the presence of an adversary that actively corrupts at most $t < n/3$ parties. Then, except with negligible probability* $\mathsf{negl}(\kappa)$*, the following properties hold:*

- ***Termination:*** *All honest parties would eventually output a set of shared randomness.*
- ***Consistency.*** *All honest parties output shares that are consistent with the same committed polynomials.*
- ***Secrecy and randomness:*** *Before reconstruction, the PPT adversary cannot predict any shared randomness better than guessing over $\mathbb{Z}_p$ (which shall hold even if all other preprocessed shared randomnesses were revealed).*

Proof. We prove each property in turn.

Termination. The termination of AsyRanGen follows from the completeness of the underlying ACSS protocol and the termination of the MVBA primitive. In the sharing phase, each party P_i invokes one ACSS instance as the dealer. By the correctness property of ACSS (Theorem 1), every instance initiated by an honest dealer eventually completes at all honest parties and outputs valid shares. Since there are at least $n-t$ honest dealers, each honest party eventually observes the completion of at least $n-t$ distinct ACSS instances, which can then be provided as input to MVBA. The parties then invoke MVBA to agree on a common set of indices corresponding to completed ACSS instances. By the validity, agreement, and termination properties of MVBA, all honest parties eventually decide on the same subset of size at least $n-t$. Since $n-t \geq 2t+1$, all honest parties can deterministically select the same canonical ordered subset of $2t+1$ completed instances from the decided set. The final extraction step applies a fixed Vandermonde transformation to the gathered shares. This extraction is a local linear computation and requires no communication. Consequently, every honest party can locally derive its output shares, establishing termination of AsyRanGen.

Consistency. By the completeness property of ACSS, every completed ACSS instance defines a unique batch of degree-t polynomials and corresponding commitments, such that the shares held by honest parties are consistent with these commitments. The MVBA protocol further guarantees that all honest parties agree on the same subset of completed ACSS instances. Since the canonical extraction subset is selected deterministically from this agreed subset, all honest parties use shares from the same family of input polynomials and the same set of input commitments at the start of extraction.

In the extraction step, each party applies the same fixed Vandermonde transformation to its local shares and combines the corresponding commitments homomorphically using the same coefficients. By linearity, the resulting shares held by all honest parties are evaluations of the same extracted degree-t polynomials, while the resulting public commitments are identical. Moreover, each extracted share verifies against the corresponding commitment. Hence, all honest parties obtain a consistent sharing of the same output values.

Secrecy and randomness. We first argue the secrecy of honest ACSS invocation. Specifically, for every ACSS instance whose dealer is honest, the secrecy of ACSS from Theorem 1 implies that the adversary learns no information about the dealer's input secrets. It remains to address the concurrent-composition attack discussed in [67], where the adversary may try to use ciphertexts from one ACSS instance as decryption queries in another instance. In our construction, each receiver uses an independent public key for each dealer's ACSS instance. Thus, if the adversary replays a ciphertext from one instance in another instance, the honest receiver decrypts it using the secret key registered for the latter instance. The resulting plaintext is independent of the original encrypted share and fails the required commitment checks except with negligible probability. Since honest parties enter share recovery only after receiving a valid share, the adversary cannot trigger recovery on honestly generated honest-to-honest ciphertexts except with negligible probability. Therefore, ciphertexts addressed to honest parties remain unopened. By the IND-CPA security of the encryption scheme, they reveal no information about honest parties' shares. Finally, MVBA only selects a common set of completed instance indices and does not reveal any information from those shares in ACSS that are invoked by honest dealers. Hence, the shares and input secrets in honest ACSS invocations remain secure from the adversary before reconstruction, except for the shares already held by corrupted parties.

We now prove randomness. Let $\mathcal{T}$ be the subset of completed ACSS instances agreed upon by MVBA, and let $\mathcal{J} = (\mathcal{J}_1, \ldots, \mathcal{J}_{2t+1})$ be the canonical ordered subset selected from

$\mathcal{T}$ for extraction. Since $|\mathcal{J}| = 2t+1$ and at most t dealers are corrupted, at least $t+1$ selected instances are initiated by honest dealers. Without loss of generality, we assume that the first $t+1$ selected instances are initiated by honest dealers. For each batch index $k \in [B]$, let $\mathbf{r}_k \in \mathbb{Z}_p^{t+1}$ denote the vector of $t+1$ random values extracted from the selected ACSS instances, and let $\mathbf{s}'_k = \left(s_k^{(\mathcal{J}_1)}, \ldots, s_k^{(\mathcal{J}_{t+1})}\right)^\top$ be the vector of secrets contributed by these honest dealers. Then the extraction can be written as $\mathbf{r}_k = \mathbf{V}'\mathbf{s}'_k + \mathbf{c}_k$, where $\mathbf{V}'$ is the $(t+1)\times(t+1)$ submatrix of $\mathbf{V}$ formed by the first $t+1$ columns, and $\mathbf{c}_k$ collects the contribution of all remaining selected instances. From the adversary's perspective, $\mathbf{c}_k$ is fixed once the remaining contributions are fixed. The $t+1$ honest dealers sample their secrets independently and uniformly, so $\mathbf{s}'_k$ is uniform over $\mathbb{Z}_p^{t+1}$. By the hyper-invertibility, $\mathbf{V}'$ is nonsingular. Hence, the function sending $\mathbf{s}'_k$ to $\mathbf{V}'\mathbf{s}'_k + \mathbf{c}_k$ is a bijection over $\mathbb{Z}_p^{t+1}$. It follows that $\mathbf{r}_k$ is jointly uniform over $\mathbb{Z}_p^{t+1}$ from the adversary's perspective. Finally, we consider secrecy when other preprocessed shared random values are revealed. By the joint uniformity just shown, revealing any t coordinates of $\mathbf{r}_k$ imposes t independent linear constraints on $\mathbf{s}'_k$, restricting it to a one-dimensional affine subspace on which it is computationally indistinguishable from a uniformly random vector from the adversary's perspective. Since the row corresponding to any unrevealed coordinate is linearly independent of the revealed rows, the unrevealed shared randomness varies over all of $\mathbb{Z}_p$ as $\mathbf{s}'_k$ ranges over that affine subspace. Hence the unrevealed randomness remains uniformly distributed over $\mathbb{Z}_p$.

Combining the above arguments, AsyRanGen satisfies the secrecy and randomness properties. □

D.7 Step 3: Asynchronous Multiplication Triple Generation using New ACSS

In this section, we elaborate on how DumboMPC++ improves DumboMPC in preprocessing the shared multiplication triples by (i) integrating our improved ACSS instantiation and (ii) adopting the batched proof for product relationship over batched polynomial commitments. The detailed construction of the asynchronous triple generation (AsyTriGen) protocol is given in Algorithm 5. It follows the BGW multiplication paradigm [71]: the participating parties locally multiply shared values, reshare the resulting degree-$2t$ product shares (with a zk-proof attesting the secret under resharing is indeed the product of two legitimate shares), and then combine valid resharings to obtain degree-t sharings of the products.

INGREDIENTS. The AsyTriGen relies on the following components.

- The ACSS protocol, described in Algorithm 4.
- The Multi-Valued Validated Asynchronous Byzantine Agreement (MVBA) protocol [34].
- Non-interactive zero-knowledge proof-of-knowledge system [5, 16] consist of
  - A non-interactive zero-knowledge proof of knowledge $\mathsf{PoK}_{\mathsf{ComEq}}$ proves that a generalized Pedersen commitment and a sequence of standard Pedersen commitments open to the same ordered vector. Let $\mathbf{X} = (\mathsf{Ped}_{x_1}, \ldots, \mathsf{Ped}_{x_B}) \in \mathbb{G}^B$, $\mathbf{x} = (x_1, \ldots, x_B) \in \mathbb{Z}_p^B$, $\hat{x} \in \mathbb{Z}_p$, and $\mathbf{r} = (r_1, \ldots, r_B) \in \mathbb{Z}_p^B$. Formally, the proof system realizes the relation

$$\{(\mathsf{pp}, Y, \mathbf{X}; \mathbf{x}, \hat{x}, \mathbf{r}) : C_x = h^{\hat{x}} \prod_{i=1}^{B} g_i^{x_i} \wedge \bigwedge_{i\in[B]} \mathsf{Ped}_{x_i} = g^{x_i} h^{r_i}\}.$$

    We use this proof to link each generalized commitment to the corresponding standard Pedersen commitments. For the sake of concise presentation, we sometimes also omit the public parameter pp from the syntax in later presentations.
  - A Bulletproof-style non-interactive zero-knowledge proof of knowledge $\mathsf{PoK}_{\mathsf{BP}}$ for proving that three sequences of standard Pedersen commitments open to vectors $\mathbf{a}, \mathbf{b}, \mathbf{c}$ satisfying $\mathbf{a} \circ \mathbf{b} = \mathbf{c}$. Let $\{\mathsf{Ped_}a_i\}_{i\in[B]}$, $\{\mathsf{Ped_}b_i\}_{i\in[B]}$, and $\{\mathsf{Ped_}c_i\}_{i\in[B]}$ be public sequences of standard Pedersen commitments. For witness $(\mathbf{a}, \mathbf{b}, \mathbf{c}, \tilde{\mathbf{a}}, \tilde{\mathbf{b}}, \tilde{\mathbf{c}})$, the relation is

$$\left\{ \begin{array}{l} (\mathsf{pp}, \{\mathsf{Ped_}a_i, \mathsf{Ped_}b_i, \mathsf{Ped_}c_i\}_{i\in[B]}; \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}, \tilde{\mathbf{a}}, \tilde{\mathbf{b}}, \tilde{\mathbf{c}}) : \\ \mathsf{Ped_}a_i = g^{a_i} h^{\tilde{a}_i} \wedge \mathsf{Ped_}b_i = g^{b_i} h^{\tilde{b}_i} \wedge \mathsf{Ped_}c_i = g^{c_i} h^{\tilde{c}_i} \\ \wedge a_i b_i = c_i, \forall i \in [B] \end{array} \right\}.$$

    This proof is instantiated by the arithmetic-circuit Bulletproof protocol of [16], specialized to the Hadamard product relation. For simplicity of presentation, we may omit the public parameter pp from the syntax in subsequent presentations.

Algorithm 5 Efficient Asynchronous Triple Generation

Input: public parameters pp
Output: $\mathsf{Beaver_triples} = \{[\![a_k]\!]^i, [\![b_k]\!]^i, [\![c_k]\!]^i\}_{k=1}^B$

1: parties joint invoke AsyRanGen to get $2B$ randomness and divide into two parts $\{C_a^i, \{[\![a_k]\!]^i\}_{k\in[B]}, [\![\hat{a}]\!]^i\}$ and $\{C_b^i, \{[\![b_k]\!]^i\}_{k\in[B]}, [\![\hat{b}]\!]^i\}$
2: **for** $k = 1$ to B
3: Sample hiding randomness $\tilde{a}_k^i, \tilde{b}_k^i$, and $\tilde{c}_k^i$
4: $[\![c_k]\!]_{2t}^i \leftarrow [\![a_k]\!]^i \cdot [\![b_k]\!]^i$
// Standard Pedersen Commitment
5: $\mathsf{Ped}_{a_k}^i \leftarrow \mathsf{Com}([\![a_k]\!]^i, \tilde{a}_k^i)$
6: $\mathsf{Ped}_{b_k}^i \leftarrow \mathsf{Com}([\![b_k]\!]^i, \tilde{b}_k^i)$
7: $\mathsf{Ped}_{c_k}^i \leftarrow \mathsf{Com}([\![c_k]\!]_{2t}^i, \tilde{c}_k^i)$
8: invoke ACSS^i to reshare $\{[\![c_k]\!]_{2t}^i\}_{k\in[B]}$ with Commitment C_c^i and hiding randomness $[\![\hat{c}^i]\!]$ ▷ Here, C_c^i is generalized Pedersen commitment to $\{[\![c_k]\!]_{2t}^i\}_{k\in[B]}$
9: **for** $z \in \{a, b, c\}$ **do**

10: $\mathsf{eqProof}_z^i \leftarrow \mathsf{PoK}_{\mathsf{ComEq}}.\mathsf{prove}\left(\begin{matrix} C_z^i, \{\mathsf{Ped}_{z_k}^i\}_{k\in[B]}; \\ \{[\![z_k]\!]^i\}_{k\in[B]}, \hat{z}^i, \{\tilde{z}_k^i\}_{k\in[B]} \end{matrix}\right)$

▷ For $z = c$, interpret $[\![z_k]\!]^i$ as $[\![c_k]\!]^i_{2t}$.

11: $\mathsf{bpProof}^i \leftarrow \mathsf{PoK}_{\mathsf{BP}}.\mathsf{prove}\left(\begin{matrix} \{\mathsf{Ped}_{a_k}^i, \mathsf{Ped}_{b_k}^i, \mathsf{Ped}_{c_k}^i\}_{k\in[B]}; \\ \{[\![a_k]\!]^i, [\![b_k]\!]^i, [\![c_k]\!]^i_{2t}, \tilde{a}_k^i, \tilde{b}_k^i, \tilde{c}_k^i\}_{k\in[B]} \end{matrix}\right)$

12: $\mathsf{RBC}\left(\begin{matrix} \mathsf{eqProof}_a^i, \mathsf{eqProof}_b^i, \mathsf{eqProof}_c^i, \\ \{\mathsf{Ped}_{a_k}^i, \mathsf{Ped}_{b_k}^i, \mathsf{Ped}_{c_k}^i\}_{k\in[B]}, \mathsf{bpProof}^i \end{matrix}\right)$

13: **while** ACSS^j outputs and MVBA does not output $\mathcal{T}$:

14: **upon** receiving $\left(\begin{matrix} \mathsf{eqProof}_a^j, \mathsf{eqProof}_b^j, \mathsf{eqProof}_c^j, \\ \{\mathsf{Ped}_{a_k}^j, \mathsf{Ped}_{b_k}^j, \mathsf{Ped}_{c_k}^j\}_{k\in[B]}, \mathsf{bpProof}^j \end{matrix}\right)$

15: $v^j \leftarrow \mathsf{PoK}_{\mathsf{BP}}.\mathsf{verify}(\{\mathsf{Ped}_{a_k}^j, \mathsf{Ped}_{b_k}^j, \mathsf{Ped}_{c_k}^j\}_{k\in[B]}; \mathsf{bpProof}^j)$

16: $v_a^j \leftarrow \mathsf{PoK}_{\mathsf{ComEq}}.\mathsf{verify}(C_a^j, \{\mathsf{Ped}_{a_k}^j\}_{k\in[B]}; \mathsf{eqProof}_a^j)$

17: $v_b^j \leftarrow \mathsf{PoK}_{\mathsf{ComEq}}.\mathsf{verify}(C_b^j, \{\mathsf{Ped}_{b_k}^j\}_{k\in[B]}; \mathsf{eqProof}_b^j)$

18: $v_c^j \leftarrow \mathsf{PoK}_{\mathsf{ComEq}}.\mathsf{verify}(C_c^j, \{\mathsf{Ped}_{c_k}^j\}_{k\in[B]}; \mathsf{eqProof}_c^j)$

19: **if** $v^j \wedge v_a^j \wedge v_b^j \wedge v_c^j = 1$ **then**

20: parse $\{[\![[\![c_k]\!]^j_{2t}]\!]^i\}_{k\in[B]}$ from ACSS^j

21: $\mathcal{T}_i \leftarrow \mathcal{T}_i \cup \{j\}$

22: **if** $|\mathcal{T}_i| \geq 2t+1$ **then**

23: call MVBA using $\mathcal{T}_i$

24: **upon** MVBA outputs a set $\mathcal{T}$

25: set $\mathsf{Beaver_triples} = \varnothing$

26: **for** $k = 1$ to B

27: Interpolate $[\![c_k]\!]^i \leftarrow \sum_{j\in\mathcal{T}} \delta_j \cdot [\![[\![c_k]\!]^j_{2t}]\!]^i$, where δ_j denotes the Lagrange interpolation coefficient at zero.

28: $\mathsf{Beaver_triples} \leftarrow \mathsf{Beaver_triples} \bigcup ([\![a_k]\!]^i, [\![b_k]\!]^i, [\![c_k]\!]^i)$

29: **return** Beaver_triples

CONSTRUCTION OF AsyTriGen. With the above ingredients, we present AsyTriGen in Algorithm 5. The protocol avoids verifying pairwise product relations in turn. Instead, it combines the Hadamard-style product relations with a random challenge and proves the resulting batched Hadamard product relation.

- *Generating shared randomness for local multiplication.* All parties first execute AsyRanGen to obtain sufficient committed shared randomness. The resulting local evaluations are partitioned into two batches, denoted by $\left(C_a^i, \{[\![a_k]\!]^i\}_{k\in[B]}, [\![\hat{a}]\!]^i\right)$ and $\left(C_b^i, \{[\![b_k]\!]^i\}_{k\in[B]}, [\![\hat{b}]\!]^i\right)$. Here, C_a^i and C_b^i denote generalized Pedersen commitments to the corresponding evaluation vectors, namely $C_a^i = h^{[\![\hat{a}]\!]^i} \prod_{k\in[B]} g_k^{[\![a_k]\!]^i}$, $C_b^i = h^{[\![\hat{b}]\!]^i} \prod_{k\in[B]} g_k^{[\![b_k]\!]^i}$. For each $k \in [B]$, party P_i then locally computes the degree-$2t$ product share $[\![c_k]\!]^i_{2t} = [\![a_k]\!]^i \cdot [\![b_k]\!]^i$.
- *Resharing product shares with a batched product proof.* Each party P_i reshares the degree-$2t$ product shares $\{[\![c_k]\!]^i_{2t}\}_{k\in[B]}$ via ACSS. To ensure that honest parties can eventually reconstruct valid degree-$2t$ product sharings, the dealer must prove that the reshared values are consistent with the local products, $[\![c_k]\!]^i_{2t} = [\![a_k]\!]^i \cdot [\![b_k]\!]^i$, for all $k \in [B]$. Instead of providing one product proof for each tuple, we use $\mathsf{PoK}_{\mathsf{BP}}$ to prove the batched Hadamard product relation. However, $\mathsf{PoK}_{\mathsf{BP}}$ is formulated over standard Pedersen commitments to the inputs and outputs, whereas ACSS relies on generalized polynomial Pedersen commitments for sharing. Thus, we require dealers to additionally provide standard commitments for $([\![a_k]\!]^i, [\![b_k]\!]^i, [\![c_k]\!]^i_{2t})$ and further leverage $\mathsf{PoK}_{\mathsf{ComEq}}$ to link the generalized commitment to standard Pedersen commitments open to the same ordered vector. After this conversion, P_i invokes $\mathsf{PoK}_{\mathsf{BP}}$ on the sequences of standard Pedersen commitments to efficiently prove the product relation $([\![a_k]\!]^i)_{k\in[B]} \circ ([\![b_k]\!]^i)_{k\in[B]} = ([\![c_k]\!]^i_{2t})_{k\in[B]}$.
- *Reconstructing degree-t sharings of the products.* The commitments and batched product proofs allow honest parties to explicitly decide whether each dealer's resharing is valid. For each $k \in [B]$, once at least $2t+1$ valid resharings $\{[\![[\![c_k]\!]_{2t}]\!]^j\}_j$ are available, these resharings uniquely determine the degree-$2t$ sharing of the product $[\![a_k]\!][\![b_k]\!]$. The parties then apply the BGW degree-reduction process to obtain a degree-t sharing $[\![c_k]\!]$ from these $2t+1$ distinct valid degree-$2t$ resharings. Consequently, the parties eventually obtain the consistent Beaver triples $\{([\![a_k]\!], [\![b_k]\!], [\![c_k]\!])\}_{k\in[B]}$, which are used in the online phase.

Communication complexity of AsyTriGen**.** The communication cost of AsyTriGen for generating B triples consists of one AsyRanGen invocation to generate $2B$ shared random values, n parallel ACSS instances to reshare the degree-$2t$ product shares, n RBC instances to disseminate proofs of size $O(B+\log B)$, and one MVBA invocation. For sufficiently large B, the $O(n^2B)$ terms from ACSS and AsyRanGen dominate the lower-order proof dissemination and agreement costs. Therefore, AsyTriGen has total communication $O(n^2B)$, yielding $O(n^2)$ amortized communication per triple.

Security of AsyTriGen. The security of AsyTriGen can be summarized by the following theorem.

Theorem 3 (Termination, Validity, Secrecy)**.** *For any adversary $\mathcal{A}$ controlling at most $t < n/3$ parties in an asynchronous network of n parties, the* AsyTriGen *protocol satisfies the following properties with overwhelming probability:*

1. ***Termination.*** *If all honest parties activate the protocol, then all of them eventually complete and output the batch of triples $\left\{([\![a_k]\!]^i, [\![b_k]\!]^i, [\![c_k]\!]^i)\right\}_{k=1}^B$.*
2. ***Validity.*** *For each $k \in [B]$, the shares $\{[\![x_k]\!]^i\}$ (for each $x \in \{a,b,c\}$) held by honest parties lie on a unique polynomial of degree up to t with a constant term $x_k \in \mathbb{Z}_p$, such that $a_k \cdot b_k \equiv c_k \pmod q$.*
3. ***Secrecy.*** *For every triple $[\![a_k]\!], [\![b_k]\!], [\![c_k]\!])$, the PPT adversary learns no information about the values a_k, b_k, and c_k, beyond the constraint $a_k \cdot b_k = c_k$, i.e., not better than guessing over $\mathbb{Z}_p^2$ (which shall hold even if all other preprocessed triples were revealed).*

Proof. **Termination.** The termination of AsyTriGen follows from the liveness of its underlying primitives: AsyRanGen, ACSS, and MVBA. First, by the termination and validity of AsyRanGen, all honest parties eventually receive their shares of the random values a_k, b_k along with the corresponding commitments. Each party then locally computes the partial product shares and invokes the reshare phase via ACSS. Since at least $n - t \geq 2t + 1$ parties are honest, at least $2t + 1$ ACSS instances are initiated by honest dealers. By the termination and correctness of ACSS, every honest party eventually receives at least $2t + 1$ valid resharings. Hence, every honest party eventually obtains a valid local input set for MVBA, consisting of dealer indices whose resharings and proofs have been accepted. The liveness of MVBA then ensures that all honest parties eventually agree on a common subset $\mathcal{T}$ of size $2t + 1$. Finally, each party performs local interpolation to reconstruct the degree-t shares $[\![c_k]\!]^i$. Since the invoked subprotocols provide eventual termination in the asynchronous model, every honest party eventually outputs the batch of B triples.

Validity. Assume that a probabilistic polynomial-time adversary $\mathcal{A}$ violates the validity of AsyTriGen with non-negligible probability. Then there is an execution in which all verification checks accept, but the honest parties output triples that are not a valid batch of Beaver triples. That is, either the sharings of a_k or b_k are not well-defined degree-t sharings, or the reconstructed sharing $[\![c_k]\!]$ does not satisfy $[\![c_k]\!] = [\![a_k]\!] \cdot [\![b_k]\!]$ for some $k \in [B]$. We show that such an adversary can be used to violate at least one security or correctness property of the underlying components.

1. *Correctness of* AsyRanGen *is violated.* If the accepted sharings of a_k or b_k do not interpolate to unique degree-t polynomials, then the correctness of AsyRanGen is violated. Indeed, AsyRanGen guarantees that all honest parties obtain committed shares lying on the same random degree-t sharings.

2. *Soundness of* $\mathsf{PoK}_{\mathsf{ComEq}}$ *is violated.* Suppose a generalized Pedersen commitment produced by AsyRanGen or ACSS encodes one ordered evaluation vector, while the corresponding sequence of standard Pedersen commitments used in the product proof encodes a different vector. Since these standard commitments are accepted only together with a valid $\mathsf{PoK}_{\mathsf{ComEq}}$, this gives an accepting proof for two inconsistent committed vectors. This violates the soundness of $\mathsf{PoK}_{\mathsf{ComEq}}$ or the binding of the underlying commitment scheme.

3. *Soundness of* $\mathsf{PoK}_{\mathsf{BP}}$ *is violated.* Suppose the standard commitments are correctly linked to the corresponding generalized commitments, but the committed values do not satisfy the product relation. Then, for some accepted dealer P_i and some $k \in [B]$, $[\![c_k]\!]^i_{2t} \neq [\![a_k]\!]^i \cdot [\![b_k]\!]^i$, while the Bulletproof verification accepts. This directly violates the soundness of $\mathsf{PoK}_{\mathsf{BP}}$ for the committed Hadamard-product relation.

4. *Correctness of* ACSS *is violated.* If the accepted resharings pass all verification checks, but the final shares of c_k fail to interpolate to the corresponding degree-t sharing of $a_k b_k$, then at least one accepted ACSS instance must have delivered shares that are inconsistent with its committed polynomial while still passing verification. This contradicts the correctness of ACSS.

Thus, any adversary that violates validity with non-negligible probability can be used to break the correctness of AsyRanGen or ACSS, the soundness of $\mathsf{PoK}_{\mathsf{ComEq}}$ or $\mathsf{PoK}_{\mathsf{BP}}$, or the binding of the underlying commitment scheme. Since all these components are secure except with negligible probability, AsyTriGen satisfies the required validity property.

Secrecy. In the following sequence of games, we show that the adversary learns no information from the asynchronous triple generation protocol beyond the inherent product relation $c_k = a_k \cdot b_k$, even if t shares are held by corrupted parties and other triples are all revealed.

Game 0. This is the real-world execution of the AsyTriGen protocol.

Game 1. For clarity of description, we focus on a single group of batch size B, where the argument for the remaining groups is identical.

In this game, the simulator generates the public parameters together with a trapdoor. It samples $\alpha_1, \ldots, \alpha_B, \tau \xleftarrow{\$} \mathbb{Z}_p^*$, sets $g_k := g^{\alpha_k}$ for each $k \in [B]$, and sets $h := g^{\tau}$. It then publishes $\mathsf{pp} = (g_1, \ldots, g_B, h)$. $\mathcal{S}$ then proceeds to run the protocol on behalf of all honest nodes. Since $\alpha_1, \ldots, \alpha_B, \tau$ are uniform over $\mathbb{Z}_p^*$, the resulting generators are distributed identically to independently sampled generators in the real execution. Hence, the adversary's view in Game 1 is statistically identical to its view in Game 0.

Game 2. In this game, the simulator replaces the honest-dealer ACSS transcripts inside AsyRanGen with simulated transcripts. Specifically, for each honest dealer P_i, the simulator samples fresh values $\tilde{s}^i_1, \ldots, \tilde{s}^i_B \xleftarrow{\$} \mathbb{Z}_p$ and interpolates degree-t polynomials $\psi^i_1(X), \ldots, \psi^i_B(X)$ such that $\psi^i_k(0) = \tilde{s}^i_k$ and $\psi^i_k(j) = [\![s^i_k]\!]^j$ for every $P_j \in \mathcal{P}_C$ and all $k \in [B]$.

Such polynomials exist because $|\mathcal{P}_C| \leq t$, and the point 0 is distinct from all party indices.

Using the trapdoor $(\boldsymbol{\alpha}, \tau)$, the simulator defines the blinding polynomial $\hat{\psi}^i(X)$ as in Eq. 4, so that the generalized Pedersen commitment vector remains unchanged. Moreover, for every corrupted party $P_j \in \mathcal{P}_C$, we have $\hat{\psi}^i(j) = \hat{\phi}^i(j)$, because $\psi^i_k(j) = \phi^i_k(j)$ for all $k \in [B]$. Thus, the commitments, message shares, and blinding shares received by corrupted parties are exactly the same as in Game 1, while the hidden honest secrets are replaced by fresh random values.

The simulator then emulates the messages delivered to corrupted parties using the simulated values

$\left\{\mathbf{C}^i, \{\psi_k^i(j)\}_{k\in[B]}, \hat{\psi}^i(j)\right\}_{j\in\mathcal{C}}$. Since the public commitments and all plaintext values delivered to corrupted parties remain unchanged, these parts of the adversary's view are identical to those in Game 1. Ciphertexts addressed to honest parties may encrypt different shares, but they are computationally indistinguishable from their counterparts in Game 1 by the IND-CPA security of the encryption scheme. Hence, the adversary's view in Game 2 is computationally indistinguishable from its view in Game 1.

Game 3. In this game, the simulator modifies the ACSS executions by replacing ciphertexts sent from honest dealers to honest receivers with encryptions of dummy messages of the same length. By the IND-CPA security of the encryption scheme, encryptions of real shares are computationally indistinguishable from encryptions of dummy messages to any adversary that does not know the corresponding honest receiver's secret key. Moreover, for an honest dealer, the complaint and recovery procedures of ACSS reveal an honest-to-honest plaintext only if the adversary can produce a valid accusation against an honestly generated ciphertext. By the correctness and soundness of the ACSS verification procedure, such an accusation is accepted only with negligible probability. This argument also covers concurrent executions. Each receiver uses specific encryption keys with different dealers, so every ciphertext is bound to a particular dealer-receiver pair. If the adversary replays a ciphertext in another instance and attempts to use the accusation procedure as a decryption query, the ciphertext is decrypted under a different secret key. The resulting plaintext is independent of the original message and fails the commitment checks except with negligible probability. Hence, a replayed ciphertext cannot lead to an accepting recovery transcript except with negligible probability.

Therefore, except with negligible probability, the replaced honest-to-honest ciphertexts are never revealed in any accepting recovery transcript. Hence, Game 3 is computationally indistinguishable from Game 2.

Game 4. In this game, the simulator emulates the resharing phase for the product shares of honest parties. For each honest party $P_j \in \mathcal{P}_{\mathcal{H}}$ and each $k \in [B]$, the simulator computes $[\![c_k]\!]_{2t}^j = [\![a_k]\!]^j \cdot [\![b_k]\!]^j$ using the simulated shares fixed in Game 2. It then simulates the ACSS resharing of the vector $\{[\![c_k]\!]_{2t}^j\}_{k\in[B]}$. Specifically, it chooses resharing polynomials $\{\psi_{c,k}^j(X)\}_{k\in[B]}$ satisfying $\psi_{c,k}^j(0) = [\![c_k]\!]_{2t}^j$ and $\psi_{c,k}^j(i) = \phi_{c,k}^j(i)$ for every corrupted party $P_i \in \mathcal{P}_{\mathcal{C}}$. Such polynomials exist because $|\mathcal{P}_{\mathcal{C}}| \le t$ and the point 0 is distinct from all party indices. Using the trapdoor $(\boldsymbol{\alpha}, \tau)$, the simulator defines the corresponding hiding polynomial $\hat{\psi}_c^j(X)$ so that the generalized Pedersen commitment C_c^j remains unchanged and is consistent with the simulated product vector.

The simulator also generates standard Pedersen commitments $\{\mathsf{Ped}_{x_k}^j = g^{[\![x_k]\!]^j} h^{\tilde{x}_k^j}\}_{x\in\{a,b,c\},k\in[B]}$, where $[\![x_k]\!]^j$ denotes $[\![a_k]\!]^j$, $[\![b_k]\!]^j$, or $[\![c_k]\!]_{2t}^j$, respectively, and the hiding values $\tilde{x}_k^j$ are sampled uniformly. These commitments are distributed exactly as in the real execution and open to values satisfying the Hadamard relation.

Instead of running the real provers for $\mathsf{PoK}_{\mathsf{ComEq}}$ and $\mathsf{PoK}_{\mathsf{BP}}$, the simulator invokes their zero-knowledge simulators:

- For each $x \in \{a,b,c\}$, it invokes $\mathcal{S}_{\mathsf{ComEq}}$ on the statement $\left(\mathsf{pp}, C_x^j, \{\mathsf{Ped}_{x_k}^j\}_{k\in[B]}\right)$, simulating a proof that the generalized commitment C_x^j and the corresponding standard Pedersen commitments open to the same ordered evaluation vector.
- It invokes $\mathcal{S}_{\mathsf{BP}}$ on the statement $\left(\mathsf{pp}, \{\mathsf{Ped}_{a,k}^j, \mathsf{Ped}_{b,k}^j, \mathsf{Ped}_{c,k}^j\}_{k\in[B]}\right)$, simulating a proof of the committed Hadamard product relation $([\![a_k]\!]^j)_{k\in[B]} \circ ([\![b_k]\!]^j)_{k\in[B]} = ([\![c_k]\!]_{2t}^j)_{k\in[B]}$.

By the hiding argument for the simulated ACSS resharings above and the zero-knowledge property of $\mathsf{PoK}_{\mathsf{ComEq}}$ and $\mathsf{PoK}_{\mathsf{BP}}$, the adversary's view in Game 4 is computationally indistinguishable from its view in Game 3.

Since each adjacent pair of games is statistically or computationally indistinguishable, the adversary's view in the real execution is indistinguishable from its view in Game 4. The view in Game 4 is generated using only the public parameters, the shares held by corrupted parties, and the prescribed multiplication relation $c_k = a_k b_k$ for the generated triples, while all proof transcripts are produced by zero-knowledge simulators. Hence, the simulated view reveals no additional information about the honest parties' shares. This proves the secrecy of AsyTriGen. □

E Basic Online-Phase Operations Supported in Robust AMPC

In this appendix, we briefly recall several standard online-phase operations provided by the existing AMPC framework [58], which we use to instantiate our DumboMix family of mixing protocols in the target asynchronous setting. Given the preprocessing materials from Appendix D.2, including shared randomness and shared multiplication triples, an AMPC framework can already support (i) robust reconstruction, (ii) multiplication of shared values, and (iii) batched reconstruction, in the fully asynchronous setting.

Robust Reconstruction. Algorithm 6 details the procedure for robustly decoding a degree-t sharing in the asynchronous setting with $t < n/3$. The algorithm operates in two stages: an optimistic check and a robust decoding fallback. Initially, upon receiving any $2t+1$ shares, the party attempts to interpolate a degree-t polynomial $\phi(\cdot)$ using an arbitrary subset of $t+1$ points. If $\phi(\cdot)$ is consistent with all $2t+1$ received shares, it is accepted as correct. Otherwise, the party continues to

collect additional shares and invokes a Reed–Solomon decoding algorithm on the current set $\mathcal{V}$ of received shares. For $|\mathcal{V}|$ received shares, the decoder can correct up to $\left\lfloor \frac{|\mathcal{V}|-t-1}{2} \right\rfloor$ erroneous shares, ensuring eventual and unique recovery once sufficiently many honest shares have been received.

Algorithm 6 RobustRec

Input: A degree-t sharing $[\![s]\!]$ distributed among n parties.
Output: The reconstructed secret s (or sometimes a t-degree polynomial whose zero-point evaluation encodes the secret s).
P_i sends $(i, [\![s]\!]^i)$ to all parties
wait until P_i receives a set of shares $\mathcal{U} = \{(j, [\![s]\!]^j)\}$ such that $|\mathcal{U}| \geq 2t+1$
  Let $\mathcal{V} \subseteq \mathcal{U}$ be the current set of received shares
  if $|\mathcal{V}| = 2t+1$
    Interpolate degree-t polynomial ϕ from any $t+1$ points in $\mathcal{V}$
    if $\phi(j) = [\![s]\!]^j$ for all $(j, [\![s]\!]^j) \in \mathcal{V}$
      return $\phi(0)$
    wait for additional shares to update $\mathcal{V}$;
    Run Reed–Solomon decoding on $\mathcal{V}$ for a degree-t polynomial
      if decoding returns a unique polynomial ϕ
        return $\phi(0)$, or sometimes **return** ϕ

Share Multiplication. We use Beaver's standard multiplication trick [10] to multiply two degree-t shared values. Given degree-t sharings $[\![x]\!]$ and $[\![y]\!]$, and a preprocessed Beaver triple $([\![a]\!], [\![b]\!], [\![c]\!])$ with $c = ab$, the parties open the masked differences $d = x - a$ and $e = y - b$ using RobustRec. Each party then locally computes

$$[\![z]\!] = de + d[\![b]\!] + e[\![a]\!] + [\![c]\!],$$

which is a degree-t sharing of $z = xy$. Since a and b are random and hidden before d and e are opened, the reconstruction of these masked differences reveals no information about x or y beyond what is leaked by the multiplication output.

Algorithm 7 Mul

Input: Shares $[\![x]\!], [\![y]\!]$ and a precomputed triple $([\![a]\!], [\![b]\!], [\![c]\!])$ where $c = ab$.
Output: $[\![z]\!]$ of degree-t, where $z = xy$
Parties locally compute $[\![d]\!] = [\![x]\!] - [\![a]\!]$ and $[\![e]\!] = [\![y]\!] - [\![b]\!]$
$d \leftarrow \mathsf{RobustRec}([\![d]\!])$
$e \leftarrow \mathsf{RobustRec}([\![e]\!])$
return $[\![z]\!] = d \cdot e + d \cdot [\![b]\!] + e \cdot [\![a]\!] + [\![c]\!]$

Batch Reconstruction. As inheriting the online-phase optimizations from hbMPC [58], we also use BatchRec (Algorithm 8) to reduce the communication overhead of batched reconstructions, a common scenario when many MPC multiplications are executed in parallel. The protocol reconstructs $t+1$ shared secrets simultaneously, with a communication cost of only $O(n^2)$. Rather than running $t+1$ independent instances of RobustRec, the parties first use a public Vandermonde matrix $V \in \mathbb{Z}_p^{n \times (t+1)}$ to linearly transform shares of $[\![x_1]\!], \ldots, [\![x_{t+1}]\!]$ into shares of new values $[\![y_1]\!], \ldots, [\![y_n]\!]$.

Each party P_j reconstructs its assigned value y_j using RobustRec and broadcasts the result. Once sufficiently many valid values y_j are available, every party decodes the Vandermonde encoding to recover the original secrets $x_1, \ldots, x_{t+1}$. This amortizes the reconstruction cost over $t+1$ secrets and yields near-optimal efficiency in the asynchronous setting.

Algorithm 8 BatchRec

Input: Shares $[\![x_1]\!], \ldots, [\![x_{t+1}]\!]$ of degree-t
Output: Secrets $x_1, \ldots, x_{t+1}$
Let $V \in \mathbb{Z}_p^{n \times (t+1)}$ be a Vandermonde matrix with $V_{i,j} = \eta_i^{j-1}$
Each party P_i locally computes $([\![y_1]\!]^i, \ldots, [\![y_n]\!]^i)^\top = V \cdot ([\![x_1]\!]^i, \ldots, [\![x_{t+1}]\!]^i)^\top$
// Here each y_i evaluates a degree-t polynomial $\phi(X) := \sum_{j=1}^{t+1} x_j X^{j-1}$ at the point η_i, i.e., $\{y_i\}$ are a set of t-degree secret shares of $\phi(X)$.
$y_i \leftarrow \mathsf{RobustRec}([\![y_i]\!])$ // Here each party multicasts $[\![y_i]\!]$
$\phi(X) \leftarrow \mathsf{RobustRec}(y_i)$ // Here each party multicasts y_i
return $\{x_1, \ldots, x_{t+1}\}$, which are coefficients of $\phi(X)$

F More Auxiliary Algorithms

F.1 Computing Powers in Constant Rounds

To avoid the $O(k)$ interaction rounds incurred by naively computing the powers $[\![m^2]\!], \ldots, [\![m^k]\!]$ sequentially during the online phase, we adopt the ComputePowers protocol from PowerMix [58]. Given a secret-shared value $[\![m]\!]$, this protocol computes the share of its powers, namely $[\![m^2]\!], \ldots, [\![m^k]\!]$ with only $O(1)$ interaction rounds. The protocol relies on the (preprocessed) shared random powers $([\![r]\!], [\![r^2]\!], \ldots, [\![r^k]\!])$ generated during preprocessing, which will soon be explained in the next subsection.

Specifically, given the preprocessing materials, the parties first reconstruct $C = \mathsf{RobustRec}([\![m]\!] - [\![r]\!])$. Then, by leveraging the identity

$$m^i r^j - r^{i+j} = (m-r) \sum_{\ell=0}^{i-1} m^{i-1-\ell} r^{j+\ell},$$

any monomial of the form $m^i r^j$ can be expressed as a linear combination of lower-degree terms and the public value C. In particular, the parties compute higher-order terms according

to the recurrence

$$[\![m^i r^j]\!] = [\![r^{i+j}]\!] + C \sum_{\ell=0}^{i-1} [\![m^{i-1-\ell} r^{j+\ell}]\!]. \tag{4}$$

By iteratively applying Eq 4, the parties can locally derive all target powers $[\![m^2]\!], \ldots, [\![m^k]\!]$. Overall, this requires only $O(k^2)$ local field operations while preserving $O(1)$ round complexity. The full procedure is given in Algorithm 9.

Algorithm 9 ComputePowers

Input: A shared secret $[\![m]\!]$; target degree k; preprocessed shared powers $([\![r]\!], [\![r^2]\!], \ldots, [\![r^k]\!])$ for a random secret r.
Output: Shares $([\![m]\!], [\![m^2]\!], [\![m^3]\!], \ldots, [\![m^k]\!])$.

$c \leftarrow \mathsf{RobustRec}([\![m]\!] - [\![r]\!])$
for $\ell = 2$ to k
  $[\![m^\ell]\!] \leftarrow c^\ell + \ell c^{\ell-1} [\![r]\!] + \cdots + \ell c [\![r^{\ell-1}]\!] + [\![r^\ell]\!]$
return $([\![m]\!], [\![m^2]\!], [\![m^3]\!], \ldots, [\![m^k]\!])$

F.2 Preprocessing of Shared Powers of a Random Value

To provide the preprocessing materials for the above protocol that computes powers within constant rounds, we now describe a simple (folklore) MPC subroutine that can preprocess the shared powers of a random value, which, given a secret-shared value $[\![\eta]\!] \in \mathbb{Z}_p$, outputs the sequence of its secret-shared powers $[\![\eta]\!], [\![\eta^2]\!], \ldots, [\![\eta^k]\!]$. The protocol computes these powers iteratively. Starting from $[\![\eta]\!]$, the parties repeatedly multiply the current power by $[\![\eta]\!]$ using the multiplication protocol Mul. The full procedure is described in Algorithm 10.

The above procedure computes the powers sequentially. When k is large, this step can be optimized using a baby-step giant-step strategy, which reduces the multiplicative depth while preserving the same output.

Algorithm 10 SharedPowers

Input: A shared value $[\![\eta]\!]$ and an integer $k \geq 1$
Output: Shared powers $[\![\eta]\!], [\![\eta^2]\!], \ldots, [\![\eta^k]\!]$

Set $[\![y_1]\!] \leftarrow [\![\eta]\!]$
for $j = 2, \ldots, k$
  $[\![y_j]\!] \leftarrow \mathsf{Mul}([\![y_{j-1}]\!], [\![w]\!])$
return $[\![y_1]\!], [\![y_2]\!], \ldots, [\![y_k]\!]$

F.3 Recovering Messages from Power Sums

Given the power sums $S_t = \sum_{j=1}^k m_j^t$ for $t = 1, \ldots, k$, the parties can locally reconstruct the monic polynomial $f(x) = \prod_{j=1}^k (x - m_j) = x^k + a_{k-1} x^{k-1} + \cdots + a_0$. Its coefficients can be computed recursively from the power sums using Newton's identities. Finally, the parties recover the messages by finding all k roots of $f(x)$ over $\mathbb{Z}_p$. The detailed construction is given in Algorithm 11.

Algorithm 11 Solve

Input: Reconstructed power sums $S_1, S_2, \ldots, S_k \in \mathbb{Z}_p$
Output: The set of recovered messages $\{m_1, m_2, \ldots, m_k\}$

Set $a_k \leftarrow 1$
for $j = 1$ **to** k **do** ▷ Recursive Coefficient Derivation
  $a_{k-j} \leftarrow -\frac{1}{j} \left(S_j + \sum_{i=1}^{j-1} a_{k-i} S_{j-i} \right)$
Let $f(x) = \sum_{i=0}^k a_i x^i$
Compute the multiset of all k roots, denoted as $R = \{x_i \in \mathbb{Z}_p \mid f(x_i) = 0\}_{i \in [k]}$
return R

F.4 Preprocessing of Shared Random Sign Bit

To generate a random shared sign bit in the secret-sharing manner, the parties jointly sample a random secret-shared value $[\![r]\!] \in \mathbb{Z}_p$, compute $c = r^2$, and reconstruct c. If $c = 0$, they resample $[\![r]\!]$ and repeat. Otherwise, because c is public, the parties can locally compute a square root $\sqrt{c} \in \mathbb{Z}_p$. Consequently, parties can obtain the share of the random bit $[\![r]\!](\sqrt{c})^{-1} \in \{+1, -1\}$.

Since r is uniformly random conditioned on $r \neq 0$, the value $r/\sqrt{c}$ is uniformly distributed over $\{+1, -1\}$. The full procedure is described in Algorithm 12.

Algorithm 12 RandBit

Input: A shared randomness $[\![r]\!]$ from AsyRanGen
Output: A uniformly random shared sign bit $[\![b]\!]$, where $b = \pm 1$

set $c = 0$
while $c = 0$
  Get a fresh $[\![r]\!]$ from AsyRanGen
  Compute $[\![c]\!] \leftarrow \mathsf{Mul}([\![r]\!], [\![r]\!])$
  Reconstruct $c \leftarrow \mathsf{RobustRec}([\![c]\!])$
Locally compute a square root $\sqrt{c} \in \mathbb{Z}_p$
return $[\![b]\!] \leftarrow (\sqrt{c})^{-1} \cdot [\![r]\!]$

G Precise Preprocessing Cost for Each Considered Mixing Method

Here we quantify and compare the precise expected numbers of Beaver triples and shared random values required by the preprocessing phase of four DumboMix variants and the other existing mixing methods, as summarized in Table 4. In all four variants of DumboMix, preprocessing materials are at least consumed by three procedures: (i) shared powers used for collision resolution, (ii) shared randomness used as masks for input submission, and (iii) shared triples/randomness used for

input validation; additionally, DumboMix2 and DumboMix2* require extra preprocessing for (iv) shared one-hot vectors and shared one-Hamming- weight vectors that are used in their optimistic-mixing procedure.

Shared-power generation for collision resolution. We first derive the expected number of multiplication triples and randomness used for the generation of shared powers.

Recall that each of the N client messages is placed independently in one of the L entries. Let K_i denote a random variable representing the number of messages placed at entry i. Thus, $K_i \sim \text{Binomial}\left(N, \frac{1}{L}\right)$. When $K_i \leq 1$, the entry can be processed directly without generating any shared powers. When $K_i \geq 2$, the servers compute the first K_i power sums and recover the messages contained in the entry using Newton's identities. Following the technique of [58], the servers compute the required powers of each client message using a preprocessed shared-random power tuple $\left(\llbracket \eta_j \rrbracket, \llbracket \eta_j^2 \rrbracket, \ldots, \llbracket \eta_j^{K_i} \rrbracket\right)$. Generating such a tuple requires one shared random value $\llbracket \eta_j \rrbracket$ and $K_i - 1$ multiplication triples for computing the higher powers. Since the algebraic-mixing is performed over N clients, an entry of load $K_i \geq 2$ consumes N shared random values and $N(K_i - 1)$ multiplication triples.

In DumboMix1 and DumboMix2, the servers generate shared powers for every entry with $K_i \geq 2$. Hence, we denote the expected number of triples and randomness consumed during shared power generation by $\mathbb{E}[\text{Tri}]$ and $\mathbb{E}[\text{Ran}]$, respectively.

Since $K_i \sim \text{Binomial}(N, 1/L)$, we have

$$\begin{aligned} \mathbb{E}[\text{Tri}] &= \sum_{i \in [L]} \mathbb{E}[\max(N(K_i - 1), 0)] \\ &= NL(\mathbb{E}[K_i] - \Pr(K_i \geq 1)) \\ &= N^2 - NL\left(1 - \left(1 - \frac{1}{L}\right)^N\right). \end{aligned} \tag{5}$$

Similarly, we have $\mathbb{E}(\text{Ran}) = N \sum_{i \in [L]} \Pr[K_i \geq 2]$.

For $L = 2.75N$, both DumboMix1 and DumboMix2 consume approximately $0.161N^2$ multiplication triples and $0.143N^2$ shared randomness for shared powers generation.

In DumboMix1* and DumboMix2*, each client additionally submits a sharing of the square of its message. Consequently, when $K_i = 2$, the servers already have the first two power sums required by Newton's identities and can directly resolve the collision. Thus, each server only needs to prepare the random powers for entries with $K_i \geq 3$. Accordingly, the expected numbers of triples and randomness required for shared power generation are given by

$$\begin{aligned} &\mathbb{E}[\text{Tri}^\star] = NL\left[\mathbb{E}[(K_i - 1)_+] - \Pr(K_i = 2)\right] \\ &= NL\left[\frac{N}{L} - \left(1 - \left(1 - \frac{1}{L}\right)^N\right) - \binom{N}{2}\frac{1}{L^2}\left(1 - \frac{1}{L}\right)^{N-2}\right]. \end{aligned} \tag{6}$$

and $\mathbb{E}(\text{Ran}^\star) = N \sum_{i \in [L]} \Pr[K_i \geq 3]$

For $L = 2.75N$, DumboMix1* and DumboMix2* consume approximately $0.035N^2$ multiplication triples and $0.017N^2$ shared random values for shared-power generation.

Input masking for client submissions. To securely collect inputs from untrusted clients, the servers preprocess a secret sharing of a random mask for each submitted field element and privately reconstruct the mask to the corresponding client. The client subtracts the reconstructed mask from its input and disseminates the resulting masked value to the servers using reliable broadcast. The servers can then combine the consistently delivered masked value with their shares of the mask to obtain a consistent secret sharing of the client input, without revealing the input itself.

In DumboMix1, each client submits two vectors of length L, and therefore consumes $2L$ preprocessed random masks. In DumboMix1*, each client additionally submits a length-L vector encoding the square of its message, increasing the per-client consumption to $3L$ masks. Across all N clients, the corresponding masking costs are $2NL$ and $3NL$ randomness, respectively. For $L = 2.75N$, these quantities become $5.5N^2$ and $8.25N^2$.

In DumboMix2 and DumboMix2*, clients use multiplicative masking rather than additive masking. Consequently, each client in DumboMix2 requires only one shared random value to mask its message, whereas each client in DumboMix2* requires two shared random values to mask both the message and its square. The resulting randomness costs are therefore N and $2N$, respectively.

Randomness and triples for input validation. To ensure the correctness of the subsequent shuffling procedure, the servers validate that each client input is well-formed.

In DumboMix1, the servers consume one shared random value to generate the public challenges used for batch validation. For each client, the one-hot well-formedness check and the position-consistency check consume two multiplication triples in total. Hence, validating all N client inputs consumes $2N$ multiplication triples and one shared random value.

In DumboMix1*, the servers additionally verify the square relation between the submitted message and message-square vectors. This requires one additional multiplication triple per client, resulting in a total of $3N$ multiplication triples and one shared random value.

In DumboMix2, the one-hot and corresponding one-sparse masked vectors are generated by the servers during preprocessing and therefore require no online input validation. In DumboMix2*, these vectors are likewise generated by the servers, but the square relation between the client-submitted message and its square must still be verified. Consequently, online validation consumes N multiplication triples.

Shared one-hot/one-Hamming-weight vectors (only used in DumboMix2 and DumboMix2*). Servers in DumboMix2 and DumboMix2* will generate a one-hot and correspond-

Table 4: The number of shared multiplication triples and shared randomness needed to preprocess for mixing a batch of N messages.

Scheme	Total triples	Total shared randomness	Resource breakdown
SwitchNet [58]	$N\log^2 N$	$N+\frac{N}{2}\log^2 N$	*Offline:* N randomness for input masking; $\frac{N}{2}\log^2 N$ randomness and $\frac{N}{2}\log^2 N$ triples for generating shared sign bits. *Online:* $\frac{N}{2}\log^2 N$ triples for switch evaluation.
PowerMix [58]	N^2-N	$2N$	*Offline:* N randomness for input masking; N randomness and (N^2-N) triples for shared-power generation. *Online:* None.
RabbitMix [24]	$N^2+1.603N^{3/2}$	$1.87N^{3/2}+1$	*Offline:* $1.87N^{3/2}$ randomness for input masking. *Online:* one random value for the public coin; $(N^2-0.267N^{3/2})$ triples for decompressing client-input shares, and $1.87N^{3/2}$ triples for input validation.
DumboMix1	$0.161N^2+2N$	$5.643N^2+1$	*Offline:* $5.5N^2$ randomness for input masking; $0.143N^2$ randomness and $0.161N^2$ triples for shared-power generation. *Online:* A random value for the public coin and $2N$ triples for input well-formedness validation.
DumboMix2	$5.661N^2+N\log L$	$0.143N^2+N+N\log L$	*Offline:* $N\log L$ randomness and $N\log L$ triples for generating shared sign bits; N randomness and $(5.5N^2)$ triples for generating one-hot and masked vectors; $0.143N^2$ randomness and $0.161N^2$ triples for shared-power generation. *Online:* None.
DumboMix1*	$0.035N^2+3N$	$8.266N^2+1$	*Offline:* $8.25N^2$ randomness for client masking; $0.016N^2$ randomness $0.035N^2$ triples for shared-power generation. *Online:* one random value for the public coin; $3N$ triples for input well-formedness and square-relation validation
DumboMix2*	$8.285N^2+N$ $+N\log L$	$0.016N^2+2N+N\log L$	*Offline:* $N\log L$ randomness and $N\log L$ triples for generating shared sign bits; $2N$ randomness and $(8.25N^2)$ triples for generating one-hot and masked vectors; and $0.016N^2$ randomness $0.035N^2$ triples for shared-power generation. *Online:* N triples for square-relation validation.

ing masked vectors in preprocessing to reduce the client-side communication. For each client, generating an L-dimensional random one-hot vector first requires $\log L$ shared random sign bits, whose generation consumes $\log L$ shared random values and $\log L$ multiplication triples. The servers then invoke the demultiplexing protocol to expand these bits into a random one-hot vector, consuming an additional $L-1$ multiplication triples. Using the shared one-hot vector, together with the shared mask and another L triples, the servers can generate a one-Hamming-weight masked vector whose nonzero entry is located at the same position as that of the one-hot vector. In DumboMix2, each client requires one random one-hot vector and one one-Hamming-weight masked vector. Hence, preprocessing for all N clients consumes $N\log L$ shared random values and $N(\log L+2L)$ multiplication triples.

In DumboMix2*, each client additionally requires an additional one-Hamming-weight mask vectors for its message square. Consequently, preprocessing for all clients consumes $N(\log L+2)$ shared random values and $N(\log L+3L)$ triples.

Algorithm 13 DumboMix1*

Input:

Public parameters: number of clients N, vector length L, and prime modulus p with $p>N$.

Client j's input: message $m^{(j)}$; secret shares $\{[\![r_i^{(j)}]\!]\}_{i\in[3L]}$ obtained in the preprocessing phase.

Server inputs: sufficient amount of shared randomness, Beaver triples, and shared powers.

Output: shuffled results msgs.

/* Executed by client j */

// Input encoding and submission

reconstructs $\{r_i^{(j)}\}_{i\in[3L]}$ from $\{[\![r_i^{(j)}]\!]\}_{i\in[3L]}$ using RobustRec

client j randomly selects an index $i\in[L]$

$\vec{I}^{(j)} \leftarrow [r_1^{(j)},\ldots,r_{i-1}^{(j)},1+r_i^{(j)},r_{i+1}^{(j)},\ldots,r_L^{(j)}]$

$\vec{M}^{(j)} \leftarrow [r_{L+1}^{(j)},\ldots,r_{L+i-1}^{(j)},m^{(j)}+r_{L+i}^{(j)},r_{L+i+1}^{(j)},\ldots,r_{2L}^{(j)}]$

$\tilde{M}^{(j)} \leftarrow [r_{2L+1}^{(j)},\ldots,r_{2L+i-1}^{(j)},(m^{(j)})^2+r_{2L+i}^{(j)},r_{2L+i+1}^{(j)},\ldots,r_{3L}^{(j)}]$

RBC $I^{(j)}$, $\vec{M}^{(j)}$, and $\tilde{M}^{(j)}$ to all servers

/* Executed by all each server */

// Input collection

initialize $\mathcal{L}^* \leftarrow \varnothing$

upon receiving $I^{(j)}$, $\bar{M}^{(j)}$, and $\tilde{M}^{(j)}$ from client j:
  $\mathcal{L}^* \leftarrow \mathcal{L}^* \cup \{j\}$
  if $|\mathcal{L}^*| = N$:
    invoke MVBA with a snapshot of $\mathcal{L}^*$ as input
wait until MVBA outputs $\mathcal{L}$, w.l.o.g., let $\mathcal{L} = \{1, \ldots, N\}$
// Input validation
the servers jointly reconstruct a random coin coin via Coin.Get()
for $j \in \mathcal{L}$ **(in parallel)**
  for $i \in [L]$
    $[\![I_i^{(j)}]\!] \leftarrow \bar{I}_i^{(j)} - [\![r_i^{(j)}]\!]$
    $[\![M_i^{(j)}]\!] \leftarrow \bar{M}_i^{(j)} - [\![r_{i+L}^{(j)}]\!]$
    $[\![\hat{M}_i^{(j)}]\!] \leftarrow \tilde{M}_i^{(j)} - [\![r_{i+2L}^{(j)}]\!]$
    derive $\hat{r}_i^{(j)}, \bar{r}_i^{(j)}, \check{r}_i^{(j)}, \bar{r}_i^{(j)}$ as $\hat{H}, \tilde{H}, \check{H}, \bar{H}$ on $(\mathsf{coin}\|i\|j)$, where $\bar{H} : \{0,1\}^* \to \mathbb{Z}_p$ is an independent random oracle
  compute $[\![\alpha^{(j)}]\!] = \sum_{i=1}^{L} [\![I_i^{(j)}]\!] - 1$
  compute $[\![w_i^{(j)}]\!] = \hat{r}_i^{(j)} \cdot [\![I_i^{(j)}]\!] + \bar{r}_i^{(j)} \cdot [\![M_i^{(j)}]\!] + \bar{r}_i^{(j)} \cdot [\![\hat{M}_i^{(j)}]\!]$
  compute $[\![\bar{w}^{(j)}]\!] = \sum_{i=1}^{L} [\![w_i^{(j)}]\!]$
  $[\![\beta^{(j)}]\!] = (\sum_{i=1}^{L} [\![w_i^{(j)}]\!] \cdot \check{r}_i^{(j)})^2 - [\![\bar{w}^{(j)}]\!] \cdot \left(\sum_{i=1}^{L} [\![w_i^{(j)}]\!] \cdot (\check{r}_i^{(j)})^2 \right)$
  compute $[\![\gamma^{(j)}]\!] = [\![m^{(j)}]\!] \cdot [\![m^{(j)}]\!] - [\![\hat{m}^{(j)}]\!]$
  open $\alpha^{(j)}$, $\beta^{(j)}$, and $\gamma^{(j)}$
  if any of $\alpha^{(j)}$, $\beta^{(j)}$, or $\gamma^{(j)}$ is nonzero
    set $\mathcal{L} \leftarrow \mathcal{L} / \{j\}$
//Optimistic mixing by random shifting then aggregating
for $j \in \mathcal{L}$
  derive offset $s^{(j)}$ as $H(\mathsf{coin}||j)$
  cyclically right-shift $[\![I^{(j)}]\!]$, $[\![M^{(j)}]\!]$, and $[\![\hat{M}^{(j)}]\!]$ by offset $s^{(j)}$
  update $[\![\mathbb{I}_i]\!] \leftarrow [\![\mathbb{I}_i]\!] + [\![I_i^{(j)}]\!]$
  update $[\![\mathbb{T}_i]\!] \leftarrow [\![\mathbb{T}_i]\!] + [\![M_i^{(j)}]\!]$
  update $[\![\mathbb{T}'_i]\!] \leftarrow [\![\mathbb{T}'_i]\!] + [\![\hat{M}_i^{(j)}]\!]$
Initialize $\mathsf{msgs} = \varnothing$
open tables $\mathbb{I}$, $\mathbb{T}$, and $\mathbb{T}'$
for each entry $i \in [L]$:
  if $\mathbb{I}_i = 1$: append $\mathbb{T}_i$ to msgs
  else if $\mathbb{I}_i = 2$: append $\mathsf{Solve}(\{\mathbb{T}_i, \mathbb{T}'_i\})$ to msgs
// Algebraic mixing only for collided entries
  else if $\mathbb{I}_i > 2$
    set $d \leftarrow \mathbb{I}_i$
    for $j \in \mathcal{L}$:
      $\{[\![\sigma_k^{(j)}]\!]\}_{k\in[d]} \leftarrow \mathsf{ComputePowers}([\![M_i^{(j)}]\!], d)$
    for all $k \in [d]$: $[\![\sigma_k]\!] \leftarrow \sum_{j\in\mathcal{L}} [\![\sigma_k^{(j)}]\!]$
    $\{\tilde{m}_j\}_{j\in[d]} \leftarrow \mathsf{Solve}(\{[\![\sigma_k]\!]\}_{k\in[d]})$
    append $\{\tilde{m}_j\}_{j\in[d]}$ to msgs
pad msgs with dummy symbols $\perp$ to length N
$\mathsf{coin}' \leftarrow \mathsf{Coin.Get}()$, $\tilde{\pi} \leftarrow H_{\mathsf{perm}}(\mathsf{coin}')$, and $\mathsf{msgs} \leftarrow \tilde{\pi}(\mathsf{msgs})$
remove all dummy symbols $\perp$ from msgs
return msgs

H Detailed Protocol Description of DumboMix1⋆ and DumboMix2⋆

We now present the full pseudocode of DumboMix1⋆ and DumboMix2⋆ for completeness. The optimizations there are simple and borrow the main idea from Blinder [4]: each client shares one additional value, namely the square of its input message, together with the original message. Compared with the base versions of DumboMix1 and DumboMix2, the main changes made in DumboMix1⋆ and DumboMix2⋆ lie in the phases of input validation (which additionally verifies the message's square) and a simpler algebraic mixing (which avoids the server-side from re-computing messages' squares).

Below we begin with the pseudocode of DumboMix1⋆. Algorithm 14 gives the pseudocode of DumboMix2⋆.

Algorithm 14 DumboMix2⋆

Input:
Public parameters: number of clients N, vector length L, and prime modulus p with $p > N$.
Client j's input: message $m^{(j)}$; secret shares $[\![u^{(j)}]\!]$ and $[\![\hat{u}^{(j)}]\!]$ obtained in the preprocessing phase, where $[\![u^{(j)}]\!] = \sum_{i\in[L]} [\![U_i^{(j)}]\!]$ and $[\![\hat{u}^{(j)}]\!] = \sum_{i\in[L]} [\![\hat{U}_i^{(j)}]\!]$.
Server inputs: A set of shared random one-hot vectors $\{[\![I^{(j)}]\!]\}$ and the corresponding shared random one-Hamming-weight vector $\{[\![U^{(j)}]\!]\}$ and $\{[\![\hat{U}^{(j)}]\!]\}$ for each client; sufficient amount of shared randomness and shared powers.
Output: shuffled results msgs.

/* Executed by client j */
reconstruct $[\![u^{(j)}]\!]$ and $[\![\hat{u}^{(j)}]\!]$ via RobustRec
if $u^{(j)} = 0$ or $\hat{u}^{(j)} = 0$
  request the servers to open $u^{(j)}$ or $\hat{u}^{(j)} = 0$ and wait for a fresh randomness
set $\bar{m}^{(j)} = m^{(j)} \cdot (u^{(j)})^{-1}$ and $\hat{m}^{(j)} = (m^{(j)})^2 \cdot (\hat{u}^{(j)})^{-1}$
RBC $\bar{m}^{(j)}$ and $\hat{m}^{(j)}$ to all servers

/* Executed by servers */
initialize $\mathcal{L}^* \leftarrow \varnothing$
upon receiving non-zero $\bar{m}^{(j)}$ and non-zero $\hat{m}^{(j)}$ from client j:
  $\mathcal{L}^* \leftarrow \mathcal{L}^* \cup \{j\}$
  if $|\mathcal{L}^*| = N$:
    invoke MVBA with a snapshot of $\mathcal{L}^*$ as input
wait until MVBA outputs $\mathcal{L}$, w.l.o.g., let $\mathcal{L} = \{1, \ldots, N\}$
for each $j \in \mathcal{L}$
  for each $i \in [L]$
    set $[\![M_i^{(j)}]\!] = \bar{m}^{(j)} \cdot [\![U_i^j]\!]$ and $[\![\hat{M}_i^{(j)}]\!] = \hat{m}^{(j)} \cdot [\![\hat{U}_i^j]\!]$
  if $\mathsf{RobustRec}((\sum_{i\in[L]} [\![M_i^{(j)}]\!])^2 - \sum_{i\in[L]} [\![\hat{M}_i^{(j)}]\!]) \neq 0 : \mathcal{L} \leftarrow \mathcal{L} / \{j\}$
for each $j \in \mathcal{L}$
  update $[\![\mathbb{I}]\!] \leftarrow [\![\mathbb{I}]\!] + [\![I^{(j)}]\!]$, $[\![\mathbb{T}]\!] \leftarrow [\![\mathbb{T}]\!] + [\![M^{(j)}]\!]$, $[\![\mathbb{T}']\!] \leftarrow [\![\mathbb{T}']\!] + [\![\hat{M}^{(j)}]\!]$
initialize $\mathsf{msgs} = \varnothing$
open tables $\mathbb{I}$, $\mathbb{T}$, and $\mathbb{T}'$
for each entry $i \in [L]$:
  if $\mathbb{I}_i = 1$: append $\mathbb{T}_i$ to msgs
  else if $\mathbb{I}_i = 2$: append $\mathsf{Solve}(\{\mathbb{T}_i, \mathbb{T}'_i\})$ to msgs
  else if $\mathbb{I}_i > 2$
    set $d \leftarrow \mathbb{I}_i$
    for $j \in \mathcal{L}$:
      $\{[\![\sigma_k^{(j)}]\!]\}_{k\in[d]} \leftarrow \mathsf{ComputePowers}([\![M_i^{(j)}]\!], d)$
    for all $k \in [d]$: $[\![\sigma_k]\!] \leftarrow \sum_{j\in\mathcal{L}} [\![\sigma_k^{(j)}]\!]$

28: $\{\tilde{m}_j\}_{j\in[d]} \leftarrow \mathsf{Solve}(\{[\![\sigma_k]\!]\}_{k\in[d]})$
29: append $\{\tilde{m}_j\}_{j\in[d]}$ to msgs
30: pad msgs with dummy symbols $\perp$ to length N
31: $\mathsf{coin}' \leftarrow \mathsf{Coin.Get}()$, $\tilde{\pi} \leftarrow H_{\mathsf{perm}}(\mathsf{coin}')$, and $\mathsf{msgs} \leftarrow \tilde{\pi}(\mathsf{msgs})$
32: remove all dummy symbols $\perp$ from msgs
33: **return** msgs

I Collision Analysis of Messages in DumboMix

Statistical Upper Bound on the Maximum Entry Load. To determine the maximum degree of shared powers that must be prepared, we derive a high-probability upper bound on the maximum entry load. Specifically, we choose K_{stat} as a statistical security parameter, targeting that

$$\Pr\left(\max_{i\in[L]} K_i > K_{\text{stat}}\right) \le \delta,$$

where δ denotes the admissible statistical failure probability, and K_i again is the random variable representing the number out of N messages that are randomly placed in the i-th entry of the L-size vector.

Recall $K_i \sim \text{Binomial}\left(N, \frac{1}{L}\right)$, where $L = cN$. Applying the Chernoff upper-tail bound in our setting for each K_i, we then have

$$\Pr(K_i > K_{\text{stat}}) \le e^{-1/c}\left(\frac{e}{c\cdot(K_{\text{stat}}+1)}\right)^{K_{\text{stat}}+1},$$

where e is the base of the natural logarithm.

Then, considering all possible $L = cN$ entries in the considered vector, we therefore have

$$\Pr\left(\max_{i\in[L]} K_i > K_{\text{stat}}\right) \le cN\cdot e^{-1/c}\left(\frac{e}{c\cdot(K_{\text{stat}}+1)}\right)^{K_{\text{stat}}+1},$$

That said, $\delta \le cNe^{-1/c}\left(\frac{e}{c\cdot(K_{\text{stat}}+1)}\right)^{K_{\text{stat}}+1}$, which for any fixed N, is a negligible function in K_{stat}. So K_{stat} can be safely viewed as our statistical security parameter (and we only preprocess shared powers up to it), since in such a case even if we omit the preprocessing of high-degree shared powers with a degree larger than K_{stat}, the statistical failure probability δ is still negligible in it (indicating that we can exponentially tune δ small enough by increasing K_{stat}).

For a concrete choice of K_{stat} in our setting where $c = 2.75$, we first specify a target statistical failure probability δ for each AAB instance, then K_{stat} can be determined as the smallest integer satisfying

$$2.75\cdot e^{-1/2.75}\cdot N\left(\frac{e}{2.75\cdot(K_{\text{stat}}+1)}\right)^{K_{\text{stat}}+1} \le \delta.$$

For example, the resulting choice of K_{stat}, for varying N and target δ, are listed as follows:

K_{stat} \ N / δ	10^3	10^4	$\cdots$	10^{10}	10^{11}	10^{12}
10^{-8}	10	11	$\cdots$	15	15	16
10^{-10}	12	12	$\cdots$	16	17	17
10^{-12}	13	14	$\cdots$	17	18	18

Note that in practice, the target statistical failure probability δ shall be determined by considering practical factors such as how frequently and how long AAB protocols will be run. For example, when $\delta = 10^{-8}$ and an AAB instance is executed every minute, a bad event (i.e., the online phase requires a shared power that was not preprocessed) is expected to occur only once every 190 years.

Lemma 1. *Suppose N messages are assigned independently and uniformly at random to a vector of size L. For each $i \in [L]$, let K_i denote the number of messages assigned to entry i. If processing an entry with K_i collisions requires $N\cdot K_i^2$ scalar field multiplications, then the expected computational complexity of algebraic mixing is*

$$\tilde{O}\left(N^2 + \frac{N^3}{L}\right).$$

Proof. Let C denote the total computational cost of recovering all colliding entries. By assumption, processing entry i incurs NK_i^2 scalar field operations. Therefore, we have $C = \sum_{i=1}^{L} NK_i^2$. Taking expectation on both sides and applying the linearity of expectation, we obtain

$$\mathbb{E}[C] = \sum_{i=1}^{L}\mathbb{E}[NK_i^2] = N\sum_{i=1}^{L}\mathbb{E}[K_i^2].$$

Since each of the N messages is assigned independently and uniformly at random to one of the L entries, the random variable K_i follows a binomial distribution: $K_i \sim \text{Bin}\left(N, \frac{1}{L}\right)$. Hence,

$$\mathbb{E}[K_i] = \frac{N}{L}, \qquad \text{Var}(K_i) = N\frac{1}{L}\left(1-\frac{1}{L}\right).$$

Using the identity

$$\mathbb{E}[K_i^2] = \text{Var}(K_i) + \left(\mathbb{E}[K_i]\right)^2,$$

we get

$$\mathbb{E}[K_i^2] = N\frac{1}{L}\left(1-\frac{1}{L}\right) + \left(\frac{N}{L}\right)^2.$$

Substituting this expression into the bound on $\mathbb{E}[C]$, we have

$$\mathbb{E}[C] = N\sum_{i=1}^{L}\left[N\frac{1}{L}\left(1-\frac{1}{L}\right) + \left(\frac{N}{L}\right)^2\right].$$

Because the summand is identical for all $i \in [L]$, this simplifies to

$$\mathbb{E}[C] = N \cdot L \cdot \left[N \frac{1}{L} \left(1 - \frac{1}{L}\right) + \left(\frac{N}{L}\right)^2 \right].$$

Expanding the above expression yields

$$\mathbb{E}[C] = N^2 \left(1 - \frac{1}{L}\right) + \frac{N^3}{L}.$$

Therefore, the expected computational cost of algebraic mixing invocations is $\mathbb{E}[C] = \tilde{O}\left(N^2 + \frac{N^3}{L}\right)$, which completes the proof. □

Lemma 2. *Suppose N messages are mapped independently and uniformly at random into a vector with $L = cN$ entries, where $c > 0$ is a constant. For each $i \in [L]$, let K_i denote the number of messages mapped to entry i. Then, the expected number of collision entries is linear in N; namely,*

$$\mathbb{E}[|\{i \in [L] : K_i \geq 2\}|] = O(N).$$

Proof. For each entry $i \in [L]$, define the indicator random variable

$$X_i = \begin{cases} 1, & \text{if } K_i \geq 2, \\ 0, & \text{otherwise.} \end{cases}$$

Then the total number of collision entries is $X = \sum_{i=1}^{L} X_i$, and we have $\mathbb{E}[X] = \sum_{i=1}^{L} \Pr[K_i \geq 2]$.

Since each of the N messages is assigned independently and uniformly to one of the $L = cN$ entries, we have $K_i \sim \mathrm{Bin}\left(N, \frac{1}{L}\right)$. Therefore, $\Pr[K_i \geq 2] \leq \mathbb{E}\left[\binom{K_i}{2}\right]$. Moreover, for a binomial random variable,

$$\mathbb{E}\left[\binom{K_i}{2}\right] = \binom{N}{2} \frac{1}{L^2}.$$

It follows that

$$\mathbb{E}[X] \leq \sum_{i=1}^{L} \binom{N}{2} \frac{1}{L^2} = L \cdot \binom{N}{2} \frac{1}{L^2} = \frac{N(N-1)}{2L}.$$

Substituting $L = cN$, we obtain

$$\mathbb{E}[X] \leq \frac{N(N-1)}{2cN} = \frac{N-1}{2c} = O(N).$$

Hence, the expected number of collision entries is linear in N. □

J Deferred Security Analysis of DumboMix

Before proving the security of DumboMix, we define an ideal functionality, $\mathcal{F}_{\mathsf{DumboMix}}$, that captures input-output unlinkability by arranging the outputs in a canonical order. We first prove that our protocol realizes $\mathcal{F}_{\mathsf{DumboMix}}$. Finally, we show how to obtain $\mathcal{F}_{\mathsf{AAB}}$ by adding a public-coin local reordering step, which ensures that the final output order is uniformly random.

Functionality $\mathcal{F}_{\mathsf{DumboMix}}$

$\mathcal{F}_{\mathsf{DumboMix}}$ supports a message space $\mathbb{M}$ that is $\mathbb{Z}_p^*$ or $\mathbb{Z}_p$. Let $\mathcal{H}$ and $\mathcal{C}$ denote the sets of honest and corrupted clients, respectively. Assume that $|\mathcal{H}| = N$ and $|\mathcal{C}| = F$. The ideal functionality $\mathcal{F}_{\mathsf{DumboMix}}$ interacts with the $N+F$ clients, n servers, and the adversary $\mathcal{S}$; it initializes an empty set $\mathcal{L}^*$, and proceeds by the following phases.

- **Input Submission**. Upon receiving a message (i, m_i) from client i for the first time:
  - $\mathcal{L}^* \leftarrow \mathcal{L}^* \cup \{i\}$ and send i to the adversary
  - internally record (i, m_i), where $m_i \in \mathbb{M} \cup \{\perp\}$
- **Input Selection**. After $|\mathcal{L}^*| \geq N$, wait for the adversary to choose a set $\mathcal{L} \subseteq \mathcal{L}^*$ s.t. $|\mathcal{L}| = N$:
  - initialized an empty list $\mathcal{X}$
  - for every $i \in \mathcal{L}$ s.t. $m_i \neq \perp$ (i.e., skip i if $m_i = \perp$):
    - sample random $tag_i \leftarrow [L]$ independently
    - if $i \in \mathcal{H}$: append $(tag_i, \perp, m_i)$ to $\mathcal{X}$
    - if $i \in \mathcal{C}$: append (tag_i, i, m_i) to $\mathcal{X}$ and send (tag_i, i, m_i) to $\mathcal{S}$
- **Mixing**. Wait for $\mathcal{S}$'s approval to execute:
  - order $\mathcal{X}$ increasingly by tag_i
  - re-rank messages in $\mathcal{X}$ with the same tag in the canonical order
  - send the resulting $\mathcal{X}$ to $\mathcal{S}$
  - let msgs be the corresponding ordered sequence obtained by retaining only the message components
- **Reveal**. Wait for $\mathcal{S}$ to instruct delivery:
  - output msgs to every server (at time specified by $\mathcal{S}$)

J.1 Security analysis for DumboMix1

Lemma 3 (Completeness of Input Formation Validation for DumboMix1)**.** *For any client j that submits vectors $I^{(j)}, M^{(j)} \in \mathbb{Z}_p^L$, assume that $\{\hat{r}_i^{(j)}, \tilde{r}_i^{(j)}, \check{r}_i^{(j)}\}_{i \in [L]}$ are common coins drawn from $\mathbb{Z}_p$. Then, if client j is honest and the following conditions hold:*

1. *$I^{(j)}$ is a one-hot vector;*
2. *the Hamming weight of $M^{(j)}$ is at most 1;*
3. *the nonzero entries of $I^{(j)}$ and $M^{(j)}$ are at the same position,*

then the values opened in Line 23 of Algorithm 1 satisfy $\alpha^{(j)} = \beta^{(j)} = 0$ with probability 1.

Proof. Let $\ell \in [L]$ be the unique index such that $I_\ell^{(j)} = 1$. Since $I^{(j)}$ is one-hot, we have $\alpha^{(j)} = \sum_{i=1}^{L} I_i^{(j)} - 1 = 0$.

By assumption, $M^{(j)}$ has Hamming weight at most 1, and its nonzero entry, if any, is also at position ℓ. Hence, for all

$i \neq \ell$, $I_i^{(j)} = M_i^{(j)} = 0$, and therefore

$$w_i^{(j)} = I_i^{(j)} \hat{r}_i^{(j)} + \tilde{r}_i^{(j)} M_i^{(j)} = 0.$$

Thus $w^{(j)}$ has at most one nonzero entry, namely $w_\ell^{(j)}$. It follows that $\bar{w}^{(j)} = w_\ell^{(j)}$, $\sum_{i=1}^L w_i^{(j)} \check{r}_i^{(j)} = w_\ell^{(j)} \check{r}_\ell^{(j)}$, and $\sum_{i=1}^L w_i^{(j)} (\check{r}_i^{(j)})^2 = w_\ell^{(j)} (\check{r}_\ell^{(j)})^2$. Substituting into the definition of $\beta^{(j)}$, we obtain

$$\beta^{(j)} = (w_\ell^{(j)} \check{r}_\ell^{(j)})^2 - w_\ell^{(j)} \left(w_\ell^{(j)} (\check{r}_\ell^{(j)})^2 \right) = 0.$$

Therefore, $\alpha^{(j)} = \beta^{(j)} = 0$ with probability 1. □

Lemma 4 (Soundness of Input Formation Validation for DumboMix1)**.** *For any client j that has submit vectors $I^{(j)}, M^{(j)} \in \mathbb{Z}_p^L$, assuming $\{\hat{r}_i^{(j)}, \tilde{r}_i^{(j)}, \check{r}_i^{(j)}\}_{k \in [L]}$ are all random coins from $\mathbb{Z}_p$ that are unpredicatable by clients at the time of submitting input vectors. Then, except with probability at most $4/|\mathbb{Z}_p|$, the event that all values opened in Line 23 of Algorithm 1 are zero implies that the following conditions hold:*

1. *$I^{(j)}$ is a one-hot vector;*
2. *the Hamming weight of each of $M^{(j)}$ is at most 1;*
3. *when $M^{(j)}$ is not all zeros (i.e., has Hamming weight one), the nonzero entries of $I^{(j)}$ and $M^{(j)}$are at the same position;*

Proof. Let $I^{(j)}, M^{(j)} \in \mathbb{Z}_p^L$ be an arbitrary, possibly malformed, submission, where $\mathbb{Z}_p$ is a prime field with $p > 3$. Let S denote the event that the submission passes validation, i.e., the protocol opens $\alpha^{(j)} = \beta^{(j)} = 0$.

Then, we define

$$c = \left| \left\{ i \in [L] : I_i^{(j)} \neq 0 \ \vee \ M_i^{(j)} \neq 0 \right\} \right|.$$

We consider two types of malformed submissions:

- **case** A: $c > 1$;
- **case** B: $c \leq 1$ and $I^{(j)}$ is not a one-hot vector.

Any malformed submission falls into $A \cup B$. Since A and B are disjoint, we have $\Pr[S \mid A \cup B] \leq \max\{\Pr[S \mid A], \Pr[S \mid B]\}$.

We first consider event B. Since $c \leq 1$ and $I^{(j)}$ is not one-hot, we have $\sum_{i \in [L]} I_i^{(j)} \neq 1$. Indeed, if $c = 0$, then the sum is 0; if $c = 1$, then the unique active entry of $I^{(j)}$, if any, is different from 1. Therefore, $\alpha^{(j)} = \sum_{i \in [L]} I_i^{(j)} - 1 \neq 0$, and hence $\Pr[S \mid B] = 0$.

It remains to bound $\Pr[S \mid A]$. Recall that $w_i^{(j)} = I_i^{(j)} \hat{r}_i^{(j)} + \tilde{r}_i^{(j)} M_i^{(j)}$ and $\bar{w}^{(j)} = \sum_{i=1}^L w_i^{(j)}$.

Under event A, the union of the supports of $I^{(j)}$ and $M^{(j)}$ has size $c > 1$. For each active index i, the value $w_i^{(j)}$ is a nonzero linear form in the independent randomness $(\hat{r}_i^{(j)}, \tilde{r}_i^{(j)})$, and is therefore uniformly distributed over $\mathbb{Z}_p$.

We further partition A into two cases:

$$A_1 = \{A \wedge \mathrm{HW}(w^{(j)}) \leq 1\}, \qquad A_2 = \{A \wedge \mathrm{HW}(w^{(j)}) > 1\}.$$

Conditioned on A, event A_1 requires at least $c-1$ of the c active coordinates of $w^{(j)}$ to vanish. By a union bound, $\Pr[A_1 \mid A] \leq \frac{c}{p^{c-1}} \leq \frac{2}{p}$, where the last inequality holds for $p > 3$ and is maximized at $c = 2$.

Now condition on A_2, so that $w^{(j)}$ has Hamming weight greater than one. For any fixed such $w^{(j)}$, the value $\beta^{(j)} = \left(\sum_{i=1}^L w_i^{(j)} \check{r}_i^{(j)} \right)^2 - \bar{w}^{(j)} \left(\sum_{i=1}^L w_i^{(j)} (\check{r}_i^{(j)})^2 \right)$ is a nonzero polynomial of total degree at most 2 in the independent randomness $\{\check{r}_i^{(j)}\}_{i \in [L]}$. By the Schwartz–Zippel lemma, $\Pr[\beta^{(j)} = 0 \mid A_2] \leq \frac{2}{p}$. Therefore,

$$\begin{aligned} \Pr[S \mid A] &\leq \Pr[A_1 \mid A] + \Pr[\beta^{(j)} = 0 \mid A_2] \\ &\leq \frac{2}{p} + \frac{2}{p} = \frac{4}{p}. \end{aligned}$$

Combining this with $\Pr[S \mid B] = 0$, we obtain $\Pr[S \mid A \cup B] \leq \frac{4}{p}$. Therefore, except with probability at most $4/p$, any submission that passes validation satisfies the required structural conditions. □

Lemma 5. *The proposed protocol* DumboMix1*, excluding the public permutation step in Lines 40–42 of Algorithm 1, can securely realize $\mathcal{F}_{\mathsf{DumboMix}}$. Moreover, the full protocol securely realizes $\mathcal{F}_{\mathsf{AAB}}$.*

Proof. The proof follows the same argument as for DumboMix1*. The only difference is that the simulator for DumboMix1* must additionally simulate the components introduced by the optimization, such as the squares of clients' messages. We omit the repeated details and refer the reader to the proof of Lemma 8 and Lemma 9. □

Lemma 6 (Completeness of Input Formation Validation for DumboMix1*)**.** *For any client j that submits vectors $I^{(j)}, M^{(j)}, \hat{M}^{(j)} \in \mathbb{Z}_p^L$, assume that $\{\hat{r}_i^{(j)}, \tilde{r}_i^{(j)}, \check{r}_i^{(j)}, \bar{r}_i^{(j)}\}_{i \in [L]}$ are common coins drawn from $\mathbb{Z}_p$. Then, if client j is honest and the following conditions hold:*

1. *$I^{(j)}$ is a one-hot vector;*
2. *both $M^{(j)}$ and $\hat{M}^{(j)}$ have Hamming weight at most 1;*
3. *the nonzero entries of $I^{(j)}$, $M^{(j)}$, and $\hat{M}^{(j)}$ are at the same position;*
4. *for the unique index i such that $I_i^{(j)} = 1$, it holds that $\hat{M}_i^{(j)} = \left(M_i^{(j)}\right)^2$,*

then the values opened in Line 25 of Algorithm 13 satisfy $\alpha^{(j)} = \beta^{(j)} = \gamma^{(j)} = 0$ with probability 1.

Proof. Let $i^\star \in [L]$ be the unique index such that $I^{(j)}_{i^\star} = 1$. Since $I^{(j)}$ is one-hot, we have $\sum_{i\in[L]} I^{(j)}_i = 1$, and therefore $\alpha^{(j)} = \sum_{i\in[L]} I^{(j)}_i - 1 = 0$.

Moreover, by Conditions (1)–(3), the vectors $I^{(j)}$, $M^{(j)}$, and $\hat{M}^{(j)}$ are all zero outside index $i^\star$. Recall that

$$[\![w^{(j)}_i]\!] = \hat{r}^{(j)}_i \cdot [\![I^{(j)}_i]\!] + \tilde{r}^{(j)}_i \cdot [\![M^{(j)}_i]\!] + \bar{r}^{(j)}_i \cdot [\![\hat{M}^{(j)}_i]\!].$$

It follows that $w^{(j)}_i = 0$ for all $i \neq i^\star$. Hence, $w^{(j)}$ has Hamming weight at most 1, and its only possible nonzero entry is at position $i^\star$.

Consequently, we have

$$\bar{w}^{(j)} = \sum_{i\in[L]} w^{(j)}_i = w^{(j)}_{i^\star}, \qquad \sum_{i\in[L]} w^{(j)}_i \check{r}^{(j)}_i = w^{(j)}_{i^\star} \check{r}^{(j)}_{i^\star},$$

and

$$\sum_{i\in[L]} w^{(j)}_i (\check{r}^{(j)}_i)^2 = w^{(j)}_{i^\star} (\check{r}^{(j)}_{i^\star})^2.$$

Substituting these identities into the definition of $\beta^{(j)}$, we obtain

$$\beta^{(j)} = \left(w^{(j)}_{i^\star} \check{r}^{(j)}_{i^\star}\right)^2 - \bar{w}^{(j)} \cdot w^{(j)}_{i^\star} (\check{r}^{(j)}_{i^\star})^2 = 0.$$

Furthermore, $m^{(j)} = \sum_{i\in[L]} M^{(j)}_i = M^{(j)}_{i^\star}$ and $\hat{m}^{(j)} = \sum_{i\in[L]} \hat{M}^{(j)}_i = \hat{M}^{(j)}_{i^\star}$. Condition (4) implies that $\hat{m}^{(j)} = \left(m^{(j)}\right)^2$. Hence, $\gamma^{(j)} = \left(m^{(j)}\right)^2 - \hat{m}^{(j)} = 0$.

All of the above equalities hold identically, for every choice of the common coins. Therefore,

$$\alpha^{(j)} = \beta^{(j)} = \gamma^{(j)} = 0$$

with probability 1 for every honest submission. □

Lemma 7 (Soundness of Input Formation Validation for DumboMix1*)**.** *For any client j that has submit vectors $I^{(j)}, M^{(j)}, \hat{M}^{(j)} \in \mathbb{Z}_p^L$, assuming $\{\hat{r}^{(j)}_i, \tilde{r}^{(j)}_i, \check{r}^{(j)}_i, \bar{r}^{(j)}_i\}_{i\in[L]}$ are all random coins from $\mathbb{Z}_p$ that are unpredicatable by clients at the time of submitting input vectors. Then, except with probability at most $4/p$, the event that all values opened in Line 25 of Algorithm 13 are zero implies that the following conditions hold:*

1. *$I^{(j)}$ is a one-hot vector;*
2. *both $M^{(j)}$ and $\hat{M}^{(j)}$ have Hamming weight at most 1;*
3. *the nonzero entries of $I^{(j)}$, $M^{(j)}$, and $\hat{M}^{(j)}$ are at the same position;*
4. *for the unique index i such that $I^{(j)}_i = 1$, it holds that $\hat{M}^{(j)}_i = \left(M^{(j)}_i\right)^2$,*

Proof. Let $(I^{(j)}, M^{(j)}, \hat{M}^{(j)}) \in (\mathbb{Z}_p^L)^3$ be an arbitrary submission, where $\mathbb{Z}_p$ is the finite field of order p and $p \neq 2$. Let S denote the event that the submission passes verification, i.e.,

$$\alpha^{(j)} = \beta^{(j)} = \gamma^{(j)} = 0.$$

Define

$$c := \left|\left\{i \in [L] : I^{(j)}_i \neq 0 \vee M^{(j)}_i \neq 0 \vee \hat{M}^{(j)}_i \neq 0\right\}\right|.$$

Consider the following events:

- A: $c > 1$;
- B: $c = 1$ and, for the unique active index i, $I^{(j)}_i \neq 1$;
- C: $c \leq 1$ and the square relation is violated, i.e., $\left(\sum_{i\in[L]} M^{(j)}_i\right)^2 \neq \sum_{i\in[L]} \hat{M}^{(j)}_i$.

All malformed submissions fall into $A \cup B \cup C$. Since A and $B \cup C$ are disjoint,

$$\begin{aligned}\Pr[S \mid A \cup B \cup C] &\leq \max\{\Pr[S \mid A], \Pr[S \mid B \cup C]\} \\ &\leq \max\{\Pr[S \mid A], \Pr[S \mid B] + \Pr[S \mid C]\}.\end{aligned} \tag{7}$$

We first bound $\Pr[S \mid A]$. As in the proof of Lemma 4, partition A into the disjoint events

$$A_1 := \{A \wedge \mathrm{HW}(w^{(j)}) \leq 1\}, \qquad A_2 := \{A \wedge \mathrm{HW}(w^{(j)}) > 1\},$$

where $\mathrm{HW}(w^{(j)})$ denotes the Hamming weight of $w^{(j)}$. Conditioned on A, the event A_1 occurs with probability at most $2/p$. Moreover, conditioned on A_2, the polynomial defining $\beta^{(j)}$ is nonzero and has total degree at most 2. Hence, by the Schwartz–Zippel lemma, $\Pr[\beta^{(j)} = 0 \mid A_2] \leq \frac{2}{p}$. Therefore,

$$\Pr[S \mid A] \leq \Pr[A_1 \mid A] + \Pr[\beta^{(j)} = 0 \mid A_2] \leq \frac{4}{p}. \tag{8}$$

Next consider event B. Since $c = 1$, there is a unique active index $i^\star$, and all other coordinates are zero. If $I^{(j)}_{i^\star} \neq 1$, then $\sum_{i\in[L]} I^{(j)}_i \neq 1$, so the one-hot check fails deterministically. Equivalently, $\alpha^{(j)} \neq 0$, and therefore

$$\Pr[S \mid B] = 0. \tag{9}$$

It remains to bound $\Pr[S \mid C]$. Under C, we have $c \leq 1$, so there are two cases.

If $c = 0$, then all entries of $I^{(j)}$, $M^{(j)}$, and $\hat{M}^{(j)}$ are zero. In this case,

$$m^{(j)} = \sum_{i\in[L]} M^{(j)}_i = 0 \qquad \text{and} \qquad \hat{m}^{(j)} = \sum_{i\in[L]} \hat{M}^{(j)}_i = 0,$$

and thus the square relation holds automatically. Hence this case cannot occur under C.

If $c = 1$, let $i^\star \in [L]$ be the unique index satisfies $I^{(j)}_i \neq 0 \vee M^{(j)}_i \neq 0 \vee \hat{M}^{(j)}_i \neq 0$. Then $m^{(j)} = \sum_{i\in[L]} M^{(j)}_i = M^{(j)}_{i^\star}$ and $\hat{m}^{(j)} = \sum_{i\in[L]} \hat{M}^{(j)}_i = \hat{M}^{(j)}_{i^\star}$. Since event C assumes that the square relation is violated, we have $\left(m^{(j)}\right)^2 \neq \hat{m}^{(j)}$. Therefore, $\gamma^{(j)} = \left(m^{(j)}\right)^2 - \hat{m}^{(j)} \neq 0$ deterministically, and so

$$\Pr[S \mid C] = 0. \tag{10}$$

Combining (7), (8), (9), and (10), we obtain

$$\Pr[S \mid A \cup B \cup C] \le \max\left\{\frac{4}{p}, 0+0\right\} = \frac{4}{p}.$$

That is, any malformed submission passes the validation with probability at most $4/p$. □

Lemma 8. *The proposed* DumboMix1* *protocol in Algorithm 13, excluding the public permutation step in Lines 46–48, securely realizes* $\mathcal{F}_{\mathsf{DumboMix}}$.

Proof. We prove the lemma via a sequence of games.

Game 0. This is the real execution of DumboMix1*.

Game 1. In this game, for each honest client j and each $k \in [3L]$, the simulator reprograms the sharing of the masking randomness while preserving the shares held by corrupted servers. Specifically, the simulator samples a fresh value $\dot{r}_k^{(j)} \xleftarrow{\$} \mathbb{Z}_p$ and defines a degree-t sharing that is consistent with the corrupted servers' shares $\left\{(i, [\![r_k^{(j)}]\!]^i)\right\}_{i \in \mathcal{P}_C}$ and has constant term $\dot{r}_k^{(j)}$. Since at most t servers are corrupted, such a degree-t sharing exists. The simulator samples it uniformly from the set of degree-t polynomials consistent with the corrupted servers' shares and the prescribed constant term.

Indistinguishability argument. By the secrecy and randomness of AsyRanGen, this reprogramming preserves the adversary's view up to computational indistinguishability. Hence, Game 1 is computationally indistinguishable from Game 0.

Game 2. In this game, the simulator replaces the masked encodings of honest clients' submissions with masked encodings of valid dummy inputs, using the programmed randomness from Game 1. Concretely, for each honest client j, the simulator samples $\iota_j \xleftarrow{\$} [L]$ and substitutes the underlying vectors with $I^{(j)} = \mathbf{e}_{\iota_j}$, $M^{(j)} = \mathbf{0}$, and $\hat{M}^{(j)} = \mathbf{0}$, where $\mathbf{e}_{\iota_j}$ denotes the one-hot vector whose ι_j-th entry is 1. It then forms the corresponding masked submission using the programmed masks from Game 1.

For every honest client, each coordinate of the masked submission is obtained by adding the corresponding underlying value to an independently uniform mask in $\mathbb{Z}_p$. Hence, each masked coordinate is itself uniformly distributed over $\mathbb{Z}_p$, independently of the value being masked. Replacing the real underlying vectors by the above dummy vectors therefore only translates uniform masks and does not change the distribution of the masked submission. Moreover, since ι_j is sampled uniformly from $[L]$, the substituted one-hot position has the same distribution as in the real execution.

Indistinguishability argument. The substituted encodings are also consistent with the subsequent input-validation procedure. In particular, for every honest client j, $I^{(j)}$ is one-hot and $M^{(j)} = \hat{M}^{(j)} = \mathbf{0}$. Therefore, for any validation challenges generated as in the real execution, $\alpha^{(j)} = \beta^{(j)} = \gamma^{(j)} = 0$, and every substituted honest submission is accepted, while the validation behavior of corrupted submissions is unchanged. Hence, up to the end of the input-validation phase, the adversary's view in Game 2 is identically distributed to its view in Game 1.

Game 3. Starting from the simulated prefix of Game 2, the simulator replaces the remainder of the execution with a simulation consistent with $\mathcal{F}_{\mathsf{DumboMix}}$.

Whenever a corrupted client j submits its masked encoding, the simulator recovers the underlying vectors $I^{(j)}, M^{(j)}, \hat{M}^{(j)}$ using the corresponding masks. If the submission is well formed, it determines the unique active index idx_j and the corresponding message $m_j = M^{(j)}_{idx_j}$, and submits (j, m_j) to $\mathcal{F}_{\mathsf{DumboMix}}$; otherwise, it submits $(j, \perp)$. Honest clients submit their inputs directly to the ideal functionality.

Once MVBA determines the mixing set $\mathcal{L}$, the simulator instructs $\mathcal{F}_{\mathsf{DumboMix}}$ to select the same set $\mathcal{L}$. For every valid corrupted client $j \in \mathcal{L} \cap \mathcal{C}$, the functionality samples $tag_j \xleftarrow{\$} [L]$ and reveals (tag_j, j, m_j) to the simulator, as prescribed by its input-selection phase.

The simulator then replaces the real common-coin reconstruction with a simulated reconstruction of a fresh uniform value coin, while preserving the shares already held by corrupted servers, using the same share-programming argument as in Game 1. Before completing the reconstruction to the adversary, for every well-formed corrupted client $j \in \mathcal{L}$ with active index idx_j, it programs $H(\mathsf{coin} \| j) := tag_j - idx_j \pmod L$. Consequently, $idx_j + H(\mathsf{coin} \| j) = tag_j \pmod L$, so the cyclic shift places the corrupted client's message at the position prescribed by the ideal functionality. Since tag_j is uniform over $[L]$, the programmed value $tag_j - idx_j \bmod L$ is also uniform over $[L]$, exactly as a fresh output of H. Moreover, H is independent of the random oracles used for input validation, and its inputs are domain-separated by the client identifier j. Since coin is hidden before reconstruction, the programmed points are fresh except with negligible probability.

The simulator executes the input-validation and cyclic-shifting procedures as in Game 2. Every substituted honest submission is well formed and therefore passes validation, while corrupted submissions are validated exactly as in the real execution.

Immediately before the aggregation values are opened, the simulator approves the mixing phase of $\mathcal{F}_{\mathsf{DumboMix}}$ and obtains the resulting tagged list, with rejected inputs omitted. It initializes $\hat{\mathbb{I}}, \hat{\mathbb{T}}, \hat{\mathbb{T}}'$ to zero and, for each tuple $(tag_i, \star, m_i)$ in the tagged list, sets $\hat{\mathbb{I}}_{tag_i} \leftarrow \hat{\mathbb{I}}_{tag_i} + 1$, $\hat{\mathbb{T}}_{tag_i} \leftarrow \hat{\mathbb{T}}_{tag_i} + m_i$, and $\hat{\mathbb{T}}'_{tag_i} \leftarrow \hat{\mathbb{T}}'_{tag_i} + m_i^2$.

For every entry k, the simulator samples uniformly from the set of degree-t sharings whose constant term is $\hat{\mathbb{I}}_k$ and whose evaluations at the corrupted servers agree with their existing aggregate shares

$$\left\{(i, [\![\mathbb{I}_k]\!]^i)\right\}_{i \in \mathcal{P}_C}.$$

Since at most t servers are corrupted, such sharings exist and

have the same distribution as fresh degree-t sharings conditioned on the corrupted servers' existing view. The simulator proceeds analogously for $\hat{\mathbb{T}}_k$ and $\hat{\mathbb{T}}'_k$. Thus, all opened aggregate values are consistent with the tagged state prescribed by $\mathcal{F}_{\mathsf{DumboMix}}$, while all shares already held by corrupted servers are preserved.

Indistinguishability argument. Conditioned on the programmed random-oracle points being fresh, the simulated transcript through input validation has the same distribution as in Game 2. The only remaining difference can occur if a malformed corrupted submission passes the real input-validation procedure, whereas the simulator has submitted $\perp$ for that client to $\mathcal{F}_{\mathsf{DumboMix}}$. By Lemma 7, this occurs with probability at most $4/p$ for each malformed corrupted submission. Hence, by a union bound over at most F corrupted submissions, the distinguishing advantage between Game 2 and Game 3 is at most $\frac{4F}{p} + \mathrm{negl}(\lambda)$.

Game 4. In this game, the simulator replaces the real power-sum computation with a simulated one consistent with the aggregate values fixed in Game 3. For each entry j, let $\mathcal{M}_j$ denote the multiset of messages assigned to that entry by the programmed aggregate vectors in Game 3. Whenever an entry contains more than two messages, the real protocol invokes ComputePowers to compute the required power sums, uses Newton identities to construct the corresponding polynomial, and recovers its roots. In Game 4, the simulator instead simulates the reconstructed power sums so that they are consistent with $\mathcal{M}_j$. All intermediate values opened during ComputePowers are masked with fresh randomness generated by AsyRanGen. By the secrecy and randomness of AsyRanGen, these masked openings hide the underlying intermediate values and can be simulated with a computationally indistinguishable distribution. For each entry j with $d_j := |\mathcal{M}_j| > 2$, and for each $k \in [d_j]$, define the target k-th power sum as $\hat{\sigma}_{j,k} = \sum_{m \in \mathcal{M}_j} m^k$. The simulator samples uniformly from the set of degree-t polynomials whose constant term is $\hat{\sigma}_{j,k}$ and whose evaluations at the corrupted servers agree with their existing shares $\{(i, [\![\sigma_{j,k}]\!]^i)\}_{i \in \mathcal{P}_C}$. Since at most t servers are corrupted, such polynomials exist, and their distribution is identical to that of a fresh degree-t sharing conditioned on the corrupted servers' existing shares. The simulator assigns the corresponding evaluations to the honest servers, so that opening the resulting sharing reconstructs to $\hat{\sigma}_{j,k}$.

Indistinguishability argument. Consequently, all reconstructed power sums are consistent with the programmed multisets $\{\mathcal{M}_j\}$, and the subsequent execution of Solve recovers exactly the intended messages. Since the simulator preserves the corrupted servers' existing shares and simulates only masked intermediate values and honest-server shares hidden from the adversary, Game 4 is computationally indistinguishable from Game 3, up to negligible distinguishing advantage.

Conclusion. Combining the above hybrid arguments, the real execution of DumboMix1* is computationally indistinguishable from the ideal execution with simulator $\mathcal{S}$ and functionality $\mathcal{F}_{\mathsf{DumboMix}}$, with distinguishing advantage at most $\frac{4F}{p} + \mathrm{negl}(\lambda)$. Hence, DumboMix1* securely realizes $\mathcal{F}_{\mathsf{DumboMix}}$. □

Lemma 9. *The proposed* DumboMix1* *protocol in Algorithm 13 securely realizes* $\mathcal{F}_{\mathsf{AAB}}$.

Proof. We prove the lemma via a sequence of games.

Game 0–Game 2. These games are identical to Games 0–2 in the proof of Lemma 8. In particular, by the end of Game 2, the masked submissions of honest clients have been replaced with masked encodings of valid dummy inputs, without changing the adversary's view. These substituted submissions remain well-formed and therefore pass the subsequent input-validation procedure.

Game 3. In this game, it follows Game 2 through input collection. From the first common-coin reconstruction onward, the simulator replaces the remaining execution with a simulation consistent with $\mathcal{F}_{\mathsf{AAB}}$.

Whenever a corrupted client j submits its masked encoding, the simulator recovers $I^{(j)}, M^{(j)}, \hat{M}^{(j)}$ using the corresponding masks. If the submission is well-formed, it determines a unique active index idx_j and message $m_j = M^{(j)}_{idx_j}$ and submits (j, m_j) to $\mathcal{F}_{\mathsf{AAB}}$; otherwise, it submits $(j, \perp)$. Honest clients submit their inputs directly to the ideal functionality.

Once MVBA determines the mixing set $\mathcal{L}$, the simulator instructs $\mathcal{F}_{\mathsf{AAB}}$ to select the same set $\mathcal{L}$ and approves its mixing phase. Let $\mathsf{Y} = (y_1, \ldots, y_q)$ be the message projection of the uniformly shuffled ideal output, where $q \le N$ is the number of selected non-$\perp$ inputs.

For each tuple in $\pi(\mathcal{X})$, viewed as a distinct message occurrence even when multiple tuples carry the same message value, the simulator independently samples a tag $tag_i \xleftarrow{\$} [L]$ and associates it with that occurrence. For a corrupted tuple (j, m_j), let tag_j denote its assigned tag. Let $\mathcal{M}_k$ denote the multiset of message occurrences assigned tag k. The simulator defines $\hat{\mathbb{I}}_k = |\mathcal{M}_k|$, $\hat{\mathbb{T}}_k = \sum_{m \in \mathcal{M}_k} m$, and $\hat{\mathbb{T}}'_k = \sum_{m \in \mathcal{M}_k} m^2$. For every $d_k := |\mathcal{M}_k| > 2$ and $\ell \in [d_k]$, it further defines $\hat{\sigma}_{k,\ell} = \sum_{m \in \mathcal{M}_k} m^\ell$.

Let $\mathsf{R} = (r_1, \ldots, r_q)$ denote the deterministic pre-permutation sequence obtained by ordering entries increasingly by tag and applying the prescribed canonical ordering within each collision. The simulator pads R with $N - q$ dummy symbols and samples $\tilde{\pi}$ uniformly among all permutations such that, after applying $\tilde{\pi}$ and removing the dummy symbols, the remaining message sequence is exactly Y.

For the following counting argument, we conceptually label the q real message occurrences, even when some occurrences carry identical message values, and erase these labels afterward. For any fixed relative ordering of the q real message occurrences, there are exactly $\binom{N}{q}(N-q)! = \frac{N!}{q!}$ permutations in the permutation space inducing that ordering after the

dummy symbols are removed. Since the ideal permutation is uniform over the $q!$ orderings of the labeled occurrences, each permutation over the N positions is selected with probability $\frac{1}{q!} \cdot \frac{1}{N!/q!} = \frac{1}{N!}$. Hence, $\tilde{\pi}$ is uniformly distributed over the permutation space, exactly as in the real execution.

The simulator next replaces the first common-coin reconstruction with a simulated reconstruction of an independently sampled uniform coin coin, consistently with the shares already held by the corrupted servers. By the secrecy and randomness of AsyRanGen and the same share-programming argument as in Game 1, the adversary's view of this simulated reconstruction is computationally indistinguishable from that in the real execution.

Before completing the reconstruction to the adversary, for every well-formed corrupted client $j \in \mathcal{L}$ with active index idx_j, the simulator programs $H(\mathsf{coin} \| j) := tag_j - idx_j \pmod{L}$. Consequently, $idx_j + H(\mathsf{coin} \| j) = tag_j \pmod{L}$, so the cyclic shift places every corrupted client's message at its assigned tag. Since tag_j is uniform over $[L]$, the programmed value is distributed exactly as a fresh output of H. Moreover, H is independent of the random oracles used for input validation. Since coin is unpredictable to the adversary before reconstruction, the programmed points are fresh except with negligible probability.

The simulator then executes input validation as in Game 2 and subsequently performs cyclic shifting as in the real protocol using the programmed offsets. At the aggregation phase, for every entry k it samples uniformly a degree-t sharing with constant term $\hat{\mathbb{I}}_k$ conditioned on the corrupted servers' existing shares, and proceeds analogously for $\hat{\mathbb{T}}_k$ and $\hat{\mathbb{T}}'_k$. Thus, the opened aggregate vectors equal the target vectors while the corrupted servers' view is preserved.

For every entry with $d_k > 2$, the simulator similarly replaces ComputePowers with a simulated execution reconstructing $\{\hat{\sigma}_{k,\ell}\}_{\ell \in [d_k]}$, with the corresponding degree-t sharings sampled conditioned on the corrupted servers' existing shares. The intermediate openings are masked by fresh preprocessing randomness and are therefore simulatable with computationally indistinguishable distributions. By the correctness of Newton reconstruction and Solve, the recovered pre-permutation sequence is exactly R.

Finally, when the real protocol would invoke its second common-coin instance for the public permutation, the simulator replaces this reconstruction with a simulated reconstruction of an independently sampled uniform coin coin', consistently with the shares already held by the corrupted servers. By the secrecy and randomness of AsyRanGen and the same share-programming argument as in Game 1, the adversary's view of this simulated reconstruction is computationally indistinguishable from that in the real execution. Before completing the reconstruction to the adversary, the simulator programs $H_{\mathsf{perm}}(\mathsf{coin}') := \tilde{\pi}$. Since $\tilde{\pi}$ is uniformly distributed over all permutations of the N positions, the programmed value has exactly the distribution of a fresh output of H_{perm}. Moreover, H_{perm} is independent of the preceding random oracles, and coin' is unpredictable to the adversary before reconstruction. Hence, the programmed point is fresh except with negligible probability.

By construction of $\tilde{\pi}$, after applying the final public permutation and removing the dummy symbols, the resulting sequence is exactly Y. Hence, the simulated protocol output coincides with the output prescribed by $\mathcal{F}_{\mathsf{AAB}}$ to deliver the same sequence at the corresponding output phase.

Indistinguishability argument. Except with negligible probability arising from the underlying preprocessing and random-oracle programming, the only remaining difference from Game 2 occurs if a malformed corrupted submission is accepted by input validation although the simulator submitted $\perp$ to $\mathcal{F}_{\mathsf{AAB}}$. By Lemma 7, this occurs with probability at most $4/p$ per malformed corrupted submission. Hence, by a union bound over at most F corrupted submissions, the distinguishing advantage is at most $\frac{4F}{p} + \mathrm{negl}(\lambda)$.

Conclusion. Game 3 is computationally indistinguishable from Game 2 and is distributed as the ideal execution with $\mathcal{F}_{\mathsf{AAB}}$. Hence, DumboMix1* securely realizes $\mathcal{F}_{\mathsf{AAB}}$. □

J.2 Security analysis for DumboMix2 and DumboMix2*

Before presenting the security proofs of DumboMix2 and DumboMix2*, we establish the security of Algorithm 2, which generates the uniform random one-hot preprocessing vector used by both protocols.

Lemma 10 (Correctness, uniformity, and Secrecy of Algorithm 2)**.** *Let $L = 2^\iota$ for some $\iota \geq 1$, and let $\{[\![b_d]\!]\}_{d=0}^{\iota-1}$ be secret sharings of independent uniformly random sign bits $b_d \in \{-1,+1\}$ over a finite field $\mathbb{Z}_p$, where p is a prime and $p \neq 2$. Then Algorithm 2 outputs a secret-shared vector $[\![I]\!]$ satisfying the following properties:*

- ***Correctness.*** *The vector I is one-hot. That is, there exists a unique index $j^\star \in \{0,\ldots,L-1\}$ such that $I_{j^\star} = 1$ and $I_j = 0$ for all $j \neq j^\star$.*
- ***Uniformity.*** *The index $j^\star$ is uniformly distributed over $\{0,\ldots,L-1\}$.*
- ***Secrecy.*** *For any adversary corrupting at most t parties, the adversary learns no additional information about the selected index $j^\star$.*

Proof. Now, we prove the three claims in turn.

Correctness. We first note that the linearity of the underlying secret-sharing scheme ensures that all local additions and scalar multiplications are evaluated correctly on secret shares. The share-multiplication subroutine Mul is instantiated using Beaver triples generated by AsyTriGen, whose validity is established in Theorem 3. Hence, it suffices to analyze the algebraic behavior of SplitNode on the underlying plaintext values.

Consider a call to SplitNode on an input $x \in \{0,1\}$ and a sign bit $b \in \{-1,+1\}$. The protocol computes $x_\ell = 2^{-1}(x - x_b)$ and $x_r = 2^{-1}(x + x_b)$. If $x = 0$, then clearly $(x_\ell, x_r) = (0,0)$. If $x = 1$, then:

$$(x_\ell, x_r) = \begin{cases} (0,1), & \text{if } b = +1, \\ (1,0), & \text{if } b = -1. \end{cases}$$

Hence, a zero node always expands into two zero children, whereas a one node always expands into exactly one child equal to 1 and one child equal to 0.

Algorithm 2 starts from the root value $T[0] = 1$. By induction on the depth of the binary tree, after each expansion, there is exactly one node with value 1, while every other node has value 0. After $\iota = \log_2 L$ levels, the tree has L leaves, exactly one of which equals 1. Since the output vector I consists of these leaf values, I is a one-hot vector.

Uniformity. At each depth $d \in \{0, \ldots, \iota - 1\}$, the random sign bit b_d determines whether the unique path proceeds to the left child or the right child. Therefore, the sequence $(b_0, \ldots, b_{\iota-1})$ uniquely determines a root-to-leaf path in the depth-ι binary tree. Conversely, each leaf corresponds to exactly one such sequence.

Since the bits $b_0, \ldots, b_{\iota-1}$ are independent and uniform, every path is chosen with probability $2^{-\iota} = 1/L$. Since the tree has exactly $L = 2^\iota$ leaves, each leaf index $j^\star \in \{0, \ldots, L-1\}$ is selected with probability $1/L$. Hence, the nonzero position of I is uniformly distributed.

Secrecy. The protocol consists of invocations of the secure multiplication subroutine Mul together with local linear operations. In each call to SplitNode, the only nontrivial cryptographic step is $[\![c]\!] = \mathsf{Mul}([\![x]\!], [\![b]\!])$, while all remaining computations are linear combinations of secret sharings and are therefore performed locally without interaction. Since Mul is securely instantiated using triples generated by AsyTriGen, and remains secure against any adversary corrupting at most $t < n/3$ parties, each invocation of SplitNode reveals no information beyond the resulting secret sharings of its outputs. By the security of Mul and standard composition, the adversary learns no additional information about the selected index $j^\star$ beyond the shares received by corrupted parties during the protocol. Since the sharing scheme is t-private and $j^\star$ is uniformly distributed, these shares reveal no information about $j^\star$. Hence, any adversary corrupting at most t parties cannot predict $j^\star$ with probability more than $1/L$ except with negligible advantage.

Therefore, Algorithm 2 securely outputs a secret sharing of a uniformly random one-hot vector. □

Theorem 4. *The proposed protocol* DumboMix2, *excluding the public permutation step, securely realizes* $\mathcal{F}_{\mathsf{DumboMix}}$. *Moreover, the full protocol securely realizes* $\mathcal{F}_{\mathsf{AAB}}$.

Proof. The protocol DumboMix2* extends DumboMix2 (Algorithm 3) by additionally maintaining a message-square vector. Accordingly, the security proof of DumboMix2 follows directly from that of DumboMix2* by omitting the simulation of the message-square vector. To avoid repetition, we refer the reader to Lemmas 11 and 12, which provide the full security analysis for DumboMix2*. □

Lemma 11. *The proposed* DumboMix2* *protocol in Algorithm 14, excluding the public permutation step in Lines 30–32, securely realizes* $\mathcal{F}_{\mathsf{DumboMix}}$.

Proof. Now, we prove the lemma by a sequence of games.

Game 0. This is the real execution of DumboMix2*.

Game 1. In this game, for each client j, the simulator replaces the preprocessing vector with a freshly sampled tuple $(\hat{I}^{(j)}, \hat{\boldsymbol{\ell}}^{(j)}, \hat{\hat{\boldsymbol{\ell}}}^{(j)})$, where $\hat{I}^{(j)}$ is a uniformly random one-hot vector, and $\hat{\boldsymbol{\ell}}^{(j)}$ and $\hat{\hat{\boldsymbol{\ell}}}^{(j)}$ are obtained by multiplying $\hat{I}^{(j)}$ by two independently sampled nonzero random field elements. The simulator then programs the honest servers' shares while preserving the shares already held by corrupted servers. Specifically, for each coordinate k, it samples uniformly from the set of degree-t sharings whose constant term is $\hat{I}_k^{(j)}$ and whose evaluations at the corrupted servers agree with their existing shares $\left\{(i, [\![I_k^{(j)}]\!]^i)\right\}_{i \in \mathcal{P}_C}$. Since $|\mathcal{P}_C| \le t$, these impose at most $t+1$ constraints at distinct evaluation points, and hence such a degree-t sharing exists. The simulator programs the sharings of $\hat{\boldsymbol{\ell}}^{(j)}$ and $\hat{\hat{\boldsymbol{\ell}}}^{(j)}$ analogously, conditioned on the corrupted servers' existing shares and their respective target values.

Using these programmed shares, the servers compute $[\![u^{(j)}]\!] = \sum_{k \in [L]} [\![\ell_k^{(j)}]\!]$, and $[\![\hat{u}^{(j)}]\!] = \sum_{k \in [L]} [\![\hat{\ell}_k^{(j)}]\!]$, and send the corresponding shares to client j.

Indistinguishability argument. By the secrecy and uniformity of Algorithm 2 and the secrecy and randomness of AsyRanGen, the unique nonzero position and the random masking values at that position remain hidden from the adversary before reconstruction. Moreover, the above conditional sampling preserves all shares already held by corrupted servers and has the same distribution as fresh degree-t sharings conditioned on those shares. Therefore, the adversary's view in Game 1 is computationally indistinguishable from its view in Game 0.

Game 2. In this game, the simulator replaces each honest client's message with the fixed value 1. Specifically, for each honest client j, it sets the submitted values to $(u^{(j)})^{-1}$ and $(\hat{u}^{(j)})^{-1}$, so that the corresponding encoded message and message-square value both equal 1.

Indistinguishability argument. Since $u^{(j)}$ and $\hat{u}^{(j)}$ are independent uniformly random elements of $\mathbb{Z}_p^*$, for any fixed $m^{(j)} \in \mathbb{Z}_p^*$, $m^{(j)}(u^{(j)})^{-1}$ and $(m^{(j)})^2(\hat{u}^{(j)})^{-1}$ are independently and uniformly distributed over $\mathbb{Z}_p^*$. Hence, replacing $m^{(j)}$ by 1 does not change the distribution of the submitted values. Moreover, the resulting encoded values satisfy the message-square consistency relation, so every substituted honest submission passes input validation exactly as in Game 1,

while the validation behavior of corrupted submissions is unchanged. Hence, up to the end of the input-validation phase, the adversary's view in Game 2 is identically distributed to its view in Game 1.

Game 3. Starting from the simulated prefix of Game 2, the simulator replaces the remainder of the execution with a simulation consistent with $\mathcal{F}_{\mathsf{DumboMix}}$.

Whenever a corrupted client j submits $\bar{m}^{(j)}$ and $\hat{m}^{(j)}$, the simulator recovers

$$m_j = \bar{m}^{(j)} u^{(j)} \qquad \text{and} \qquad m'_j = \hat{m}^{(j)} \hat{u}^{(j)}.$$

If $m'_j = m_j^2$, the simulator submits (j, m_j) to $\mathcal{F}_{\mathsf{DumboMix}}$; otherwise, it submits $(j, \perp)$. Honest clients submit their inputs directly to the ideal functionality.

Once MVBA determines the selection set $\mathcal{L}$, the simulator instructs $\mathcal{F}_{\mathsf{DumboMix}}$ to use the same set $\mathcal{L}$.

For each selected valid client j, let pos_j denote the unique nonzero position of the programmed one-hot vector $\hat{I}^{(j)}$ fixed in Game 1. The values $\{pos_j\}_{j\in\mathcal{L}}$ are independently and uniformly distributed over $[L]$. We therefore couple the tags sampled by $\mathcal{F}_{\mathsf{DumboMix}}$ with these positions by setting $tag_j := pos_j$. This coupling does not change the distribution of the ideal functionality.

When the functionality reaches its mixing phase, the simulator obtains the resulting tagged list and defines, for every $k \in [L]$, $\hat{\mathbb{I}}_k = |\{j : tag_j = k\}|$, $\hat{\mathbb{T}}_k = \sum_{j:tag_j=k} m_j$, and $\hat{\mathbb{T}}'_k = \sum_{j:tag_j=k} m_j^2$.

For each entry k, the simulator samples uniformly from the set of degree-t sharings whose constant term is $\hat{\mathbb{I}}_k$ and whose evaluations at the corrupted servers agree with their existing aggregate shares $\left\{(i, [\![\mathbb{I}_k]\!]^i)\right\}_{i\in\mathcal{P}_C}$. Since at most t servers are corrupted, such sharings exist, and their distribution is identical to that of a fresh degree-t sharing conditioned on the corrupted servers' existing shares. The simulator proceeds analogously for $\hat{\mathbb{T}}_k$ and $\hat{\mathbb{T}}'_k$.

Indistinguishability argument. the opened aggregate vectors are consistent with the tagged state prescribed by $\mathcal{F}_{\mathsf{DumboMix}}$, while the view already held by corrupted servers is preserved. Hence, Game 3 is computationally indistinguishable from Game 2.

Game 4. This game replaces the real power-sum computation with a simulated one consistent with the bucket multisets fixed in Game 3. For every entry k with $d_k > 2$, the simulator defines $\hat{\sigma}_{k,\ell} = \sum_{m\in\mathcal{M}_k} m^\ell$, and simulates ComputePowers so that the reconstructed power sums equal these target values. The corresponding degree-t sharings are sampled conditioned on the corrupted servers' existing shares.

The intermediate openings generated by ComputePowers are masked by fresh preprocessing randomness and can therefore be simulated with a computationally indistinguishable distribution. By the correctness of Newton reconstruction and Solve, the recovered messages are exactly those prescribed by $\mathcal{F}_{\mathsf{DumboMix}}$.

Indistinguishability argument. Hence, Game 4 is computationally indistinguishable from Game 3 and is distributed as the ideal execution with $\mathcal{F}_{\mathsf{DumboMix}}$.

Conclusion. Since Game 4 corresponds to the ideal execution with simulator $\mathcal{S}$ interacting with $\mathcal{F}_{\mathsf{DumboMix}}$, the real execution of DumboMix2* is computationally indistinguishable from the ideal execution. □

Lemma 12. *The proposed* DumboMix2* *protocol in Algorithm 14 securely realizes* $\mathcal{F}_{\mathsf{AAB}}$.

Proof. We prove the lemma via a sequence of games.

Game 0–Game 2. These games are identical to Games 0–2 in the proof of Lemma 11.

Game 3. Starting from the simulated prefix of Game 2, the simulator replaces the remainder of the execution with a simulation consistent with $\mathcal{F}_{\mathsf{AAB}}$.

Whenever a corrupted client j submits $\bar{m}^{(j)}$ and $\hat{m}^{(j)}$, the simulator determines the underlying encoded values $m_j := \sum_{i\in[L]} M_i^{(j)}$, and $m'_j := \sum_{i\in[L]} \hat{M}_i^{(j)}$. If $m'_j = m_j^2$, it submits (j, m_j) to $\mathcal{F}_{\mathsf{AAB}}$; otherwise, it submits $(j, \perp)$. Honest clients submit their inputs directly to the ideal functionality.

Once MVBA determines the selection set $\mathcal{L}$, the simulator instructs $\mathcal{F}_{\mathsf{AAB}}$ to use the same set $\mathcal{L}$ and approves its mixing phase. Let $\mathsf{Y} = (y_1, \ldots, y_q)$ be the message projection of the resulting uniformly shuffled ideal output, where $q \le N$ is the number of selected valid inputs.

For each selected valid message occurrence, the simulator independently samples $tag_i \xleftarrow{\$} [L]$. These tags have the same balls-into-bins distribution as the random one-hot positions in the real protocol. Let $\mathcal{M}_k$ denote the multiset of messages assigned to entry k. The simulator defines $\hat{\mathbb{I}}_k = |\mathcal{M}_k|$, $\hat{\mathbb{T}}_k = \sum_{m\in\mathcal{M}_k} m$, $\hat{\mathbb{T}}'_k = \sum_{m\in\mathcal{M}_k} m^2$.

For every entry k, the simulator samples uniformly a degree-t sharing whose constant term is $\hat{\mathbb{I}}_k$, conditioned on the shares already held by the corrupted servers, and proceeds analogously for $\hat{\mathbb{T}}_k$ and $\hat{\mathbb{T}}'_k$.

Indistinguishability argument. Since at most t servers are corrupted, the resulting sharings have the same distribution as fresh sharings conditioned on the corrupted servers' existing view. Hence, the opened aggregate vectors are consistent with the bucket assignment above while preserving the adversary's view. Therefore, Game 3 is computationally indistinguishable from Game 2.

Game 4. This game replaces the real power-sum computation and final public permutation with simulations consistent with the ideal output Y fixed in Game 3.

For every entry k with $d_k := |\mathcal{M}_k| > 2$, the simulator defines $\hat{\sigma}_{k,\ell} = \sum_{m\in\mathcal{M}_k} m^\ell$, and simulates ComputePowers so that these values are reconstructed. The corresponding degree-t sharings are sampled conditioned on the corrupted servers' existing shares. The intermediate openings are masked by fresh preprocessing randomness and can therefore be simulated with a computationally indistinguishable distribution. By the

correctness of Newton reconstruction and Solve, this produces a deterministic pre-permutation sequence $\mathsf{R} = (r_1, \ldots, r_q)$ containing exactly the messages specified by the bucket multisets $\{\mathcal{M}_k\}_{k\in[L]}$.

The simulator pads R with $N-q$ dummy symbols and samples $\tilde{\pi}$ uniformly among all permutations such that, after applying $\tilde{\pi}$ and removing the dummy symbols, the remaining sequence is exactly Y. For any fixed relative ordering of the q message positions, there are exactly $\binom{N}{q}(N-q)! = \frac{N!}{q!}$ permutations in the permutation space inducing that ordering. Since the ideal permutation is uniform over the $q!$ orderings of the labeled occurrences, each permutation over the N positions is selected with probability $\frac{1}{q!} \cdot \frac{1}{N!/q!} = \frac{1}{N!}$. Thus, $\tilde{\pi}$ is uniformly distributed over the permutation space, exactly as the public permutation in the real protocol.

The simulator then replaces the real final common-coin reconstruction with a simulated reconstruction of an independently sampled uniform coin coin', consistently with the shares already held by the corrupted servers. By the secrecy and randomness of AsyRanGen and the same share-programming argument as in Game 1, the adversary's view of this simulated reconstruction is computationally indistinguishable from that in the real execution. Before completing the simulated reconstruction to the adversary, it programs $H_{\mathsf{perm}}(\mathsf{coin}') := \tilde{\pi}$. Since H_{perm} is independent of the preceding random oracles and coin' is unpredictable to the adversary before reconstruction, the programmed point is fresh except with negligible probability.

Indistinguishability argument. By construction, applying the programmed public permutation and removing the dummy symbols yields exactly Y. Hence, Game 4 is computationally indistinguishable from Game 3 and is distributed as the ideal execution with simulator $\mathcal{S}$ interacting with $\mathcal{F}_{\mathsf{AAB}}$.

This concludes the proof. □

K Additional Experiments

This appendix includes additional experimental results for completeness.

K.1 Online-Phase Performance of DumboMix Variants

We first conduct an ablation study over the four variants of DumboMix introduced above to discuss the impact of each design choice and optimization.

Comparison among DumboMix variants. We evaluate the runtime of four DumboMix variants. For DumboMix2 and DumboMix2*, we set $c = 4$, since uniformity requires the shared one-hot vector length to be a power of two.

Figure 8 demonstrates the runtime versus the number of parties for $N = 2048$ inputs. When the number of servers increases from 4 to 31, all four variants exhibit only modest runtime variation, indicating good scalability and limited WAN overhead. Figure 9 shows runtime versus the number of inputs. For $N = 2048$, DumboMix1 completes in about 22 seconds, while DumboMix2 is roughly 10% faster. This improvement comes from removing the shifting and input-validation steps, which take $O(N)$ scalar multiplications to extend each client input into a vector. Under $L = 2.75N$, DumboMix1* reduces runtime to about 15 seconds by having clients attach the squares of their messages, which decreases the number of algebraic-mixing invocations. By combining both optimizations, DumboMix2* achieves the best performance, mixing 2048 messages in approximately 10 seconds.

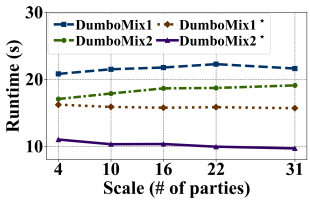


Figure 8: Online-phase runtime vs. servers # n in WAN (among DumboMix variants)

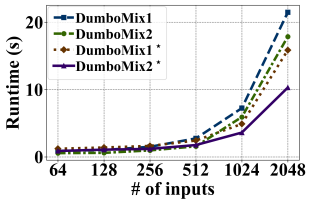
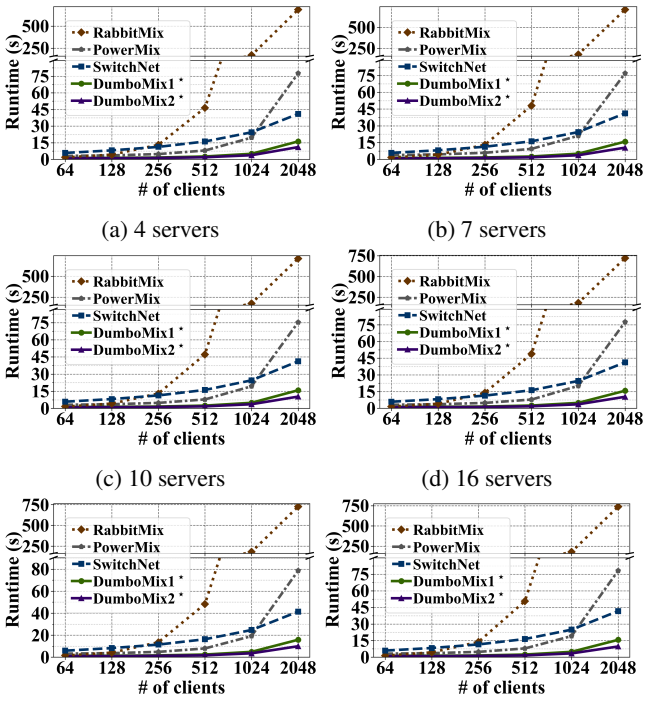


Figure 9: Online-phase runtime vs. clients # N in WAN (among DumboMix variants)

K.2 Online-Phase Performance vs. Existing Robust AABs

Runtime comparison with existing robust AAB schemes. We now present the experimental results under WAN settings, as shown in Fig. 10. We compare DumboMix1* and DumboMix2* with existing approaches across a wide range of configurations, including server scales of $n \in \{4, 7, 10, 16, 22, 31\}$ and client scales of $N \in \{64, 128, 256, 512, 1024, 2048\}$.

Increasing the number of servers in the MPC framework results in only a modest increase in runtime. In contrast, scaling the number of clients has a much stronger impact. RabbitMix becomes increasingly inefficient as the client scale grows. PowerMix also incurs rapidly growing computation costs, especially when $N \geq 1024$, due to its cubic complexity in the number of clients. SwitchNet scales more smoothly than RabbitMix and PowerMix, but its runtime remains consistently above that of DumboMix1* and DumboMix2* throughout the evaluated parameter range. By comparison, DumboMix1* and DumboMix2* scale more gracefully and consistently maintain clear advantages over the baselines.



(a) 4 servers (b) 7 servers

(c) 10 servers (d) 16 servers

(e) 22 servers (f) 31 servers

Figure 10: Online-phase runtime compared with existing AAB approaches under varying numbers of servers (in WAN).

Runtime comparison with existing robust AAB schemes under varying numbers of servers. We further evaluate the impact of server scale on runtime performance compared with SwitchNet [58], PowerMix [58], and RabbitMix [24]. Figures 11 and 12 report runtime measurements for 1024 inputs under varying server sizes in LAN and WAN environments, respectively. The results show that runtime remains largely stable as the number of servers increases.

Another observation is that SwitchNet outperforms PowerMix in LAN settings but becomes slower under WAN conditions. This performance degradation stems from SwitchNet requiring $\log^2 N$ online rounds and $O(N \log^2 N)$ MPC multiplications, which are more sensitive to network latency. In contrast, DumboMix consistently achieves strong performance across both LAN and WAN environments.

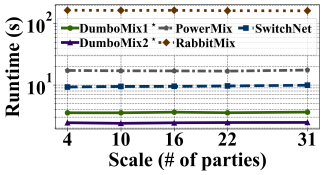


Figure 11: Online-phase runtime vs. servers # n (in LAN) compared with existing AABs

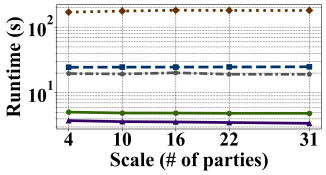


Figure 12: Online-phase runtime v.s. servers # n (in WAN) compared with existing AABs

K.3 Online-Phase Performance vs. Blinder (non-robust)

We now compare our schemes with Blinder. Unlike our security guarantees, Blinder does not provide the guarantee of robust delivery, and incurs a non-negligible chance of allowing cross-epoch linkability in asynchronous settings (cf. a concrete attacker to Blinder in Appendix C). We implement Blinder1, in which each client sends $O(N)$ messages without compaction. We exclude the RabbitMix-based compression variant because it requires $O(N^2)$ MPC multiplications, which substantially hurts performance. Following the recommendation of Blinder, we set $L = 2.75N$ in the comparison, although this configuration guarantees only 95% output delivery. Increasing the delivery probability to 99.9% requires $L = 22.02N$, resulting in approximately 9× higher overhead than the recommended setting, while still leaving a non-negligible chance of cross-epoch linkability.

We evaluate all schemes under WAN settings. Figure 13 shows that Blinder performs better than DumboMix1, DumboMix1⋆, and DumboMix2, when $c = 2.75$. Nevertheless, DumboMix1⋆ is only about 30% slower than Blinder in the WAN setting while providing full message delivery. Furthermore, DumboMix2⋆ even outperforms Blinder (though slightly), achieving an improvement of roughly 10% and simultaneously ensuring robust delivery (which Blinder lacks).

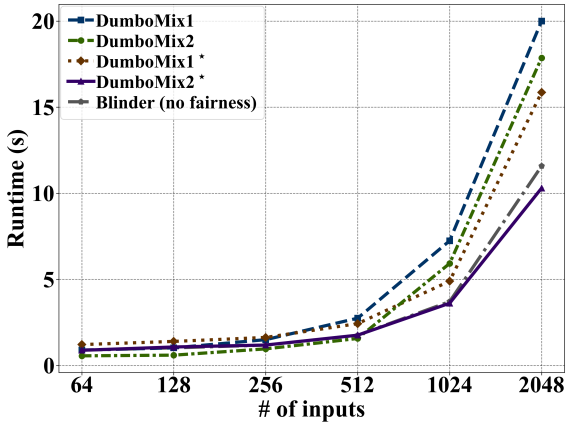


Figure 13: Online-phase runtime: ours vs. BLINDER [4] (WAN, $n = 10$)

Compared to Blinder, DumboMix takes only a marginal performance sacrifice and achieves qualitatively stronger security in terms of robustness guarantee. In particular, the online phase of DumboMix2⋆ could be even faster than that of Blinder's AMPC version in certain tests.

K.4 Offline-Phase Performance

Batch size. As in DumboMPC [67], the batch size determines how many secrets are processed within a single ACSS instance and directly affects performance. A large batch

size will amortize fixed cryptographic and communication overheads, but the gains are eventually limited by computation components and latency. In particular, we leverages Bulletproofs to accelerate multiplication proofs by aggregating B relations into a single inner-product argument of size $O(\log B)$. Thus, we evaluate batch sizes (approximately powers of two) of 10k, 30k, 60k, and 100k under WAN settings. Figure 14 shows that our throughput increases with the batch size and peaks around 60k–100k, whereas DumboMPC quickly plateaus. We therefore fix the batch size to 60k for the remaining experiments, balancing throughput, stability, and memory consumption.

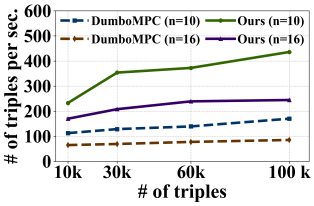


Figure 14: Triple throughput vs. batch size (WAN) in the offline phase

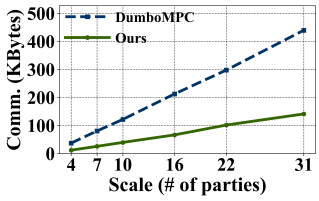


Figure 15: Amortized per-triple communication cost per server in the offline phase.

Bandwidth usage. Figure 15 reports the amortized communication cost per triple per server. Our construction reduces the communication in the ACSS phase by using generalized Pedersen commitments, which commit to an entire batch with a single group element. The remaining linear cost comes from $\mathsf{PoK}_{\mathsf{BP}}$ and $\mathsf{PoK}_{\mathsf{ComEq}}$, which together require approximately 4 group elements and 7 field elements per triple, compared with 6 group elements and 8 field elements for the product-relation proof in DumboMPC.

Since BGW-style degree reduction requires every party to reshare its local product shares together with the corresponding proof of knowledge, this per-triple proof-size reduction is amplified as the number of servers grows. Overall, for $n = 31$, the optimized preprocessing reduces the amortized communication by about $2\times$ compared with DumboMPC, as shown in Figure 15.

Offline throughput. Finally, Fig. 16 and 17 report the throughput of triple generation in both DumboMPC++ and DumboMPC for varying n of servers, in LAN and WAN settings, respectively. Across all tested numbers of servers, DumboMPC++ consistently realizes higher throughput than DumboMPC, achieving about 2-2.5 $\times$ speedup in LAN and up to 3.2 $\times$ speedup in WAN. Further offline-phase experiments (e.g., the choice of batch size) are deferred to Appendix K.4.

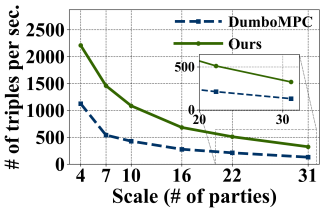


Figure 16: Triple throughput (offline) vs. servers number n in LAN setting

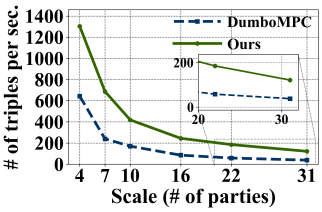


Figure 17: Triple throughput (offline) vs. servers number n in WAN setting

K.5 Client-Side Performance

Since DumboMix1* imposes the highest client-side workload among our variants, we evaluate it as a conservative benchmark for client-side deployability. Our results show that DumboMix1* remains practical.

Under the MPCaaS paradigm, the servers distribute random shares to clients for input masking. In DumboMix1$*$, each client submits three masked vectors, each of length $L = 2.75N$. To prepare these inputs, the client must reconstruct the masking randomness via dense interpolation, mask its messages, encode the resulting vectors, and compute a Merkle tree for commitment and broadcast. These steps dominate the client-side cost.

We implement the client in a virtualized Android environment on ARM-based hardware with four ARM Cortex-A78 vCPUs (ARMv8-A, 64-bit). Since the guest OS does not expose CPU frequency information, *i.e.*, no cpufreq/DVFS interface is available, we do not report clock frequency. Instead, we report Geekbench scores as a proxy for computational capability. The cloud-phone environment achieves a single-core score of 1439 and a multi-core score of 4299, with the single-core score being roughly comparable to a Snapdragon 8 Gen 1 core at 1.8 GHz.

Client-side execution time. Figure 18 reports the client-side runtime of DumboMix for different input scales, and server counts n. As N increases, the runtime rises as the client must process longer vectors and perform several expensive operations: dense interpolation for recovering masking randomness, Reed–Solomon encoding of masked vectors, and Merkle tree construction for commitment and broadcast. In our implementation, interpolation and Reed–Solomon encoding dominate the runtime and scale approximately quadratically with the number of servers, while Merkle tree construction adds further but typically smaller overhead. For a fixed N, increasing n also increases the client workload, leading to higher runtime. Hence, the configuration with the largest number of servers (e.g., $n = 31$) consistently yields the highest runtime across all tested input scales.

Nevertheless, the measured overhead remains practical. Even for the largest evaluated configuration, $N = 2048$ and $n = 31$, the client completes all operations in about 530 ms,

indicating that DumboMix is practical for commodity mobile-class devices.

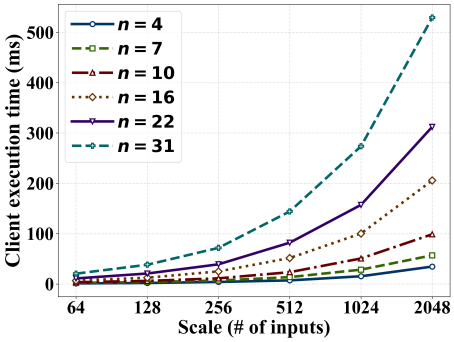


Figure 18: Client runtime of DumboMix for varying N and n

Client-side communication. Table 5 shows that the total communication cost increases almost linearly with the input scale, suggesting that the dominant communication term is proportional to the vector length. Although increasing the number of servers introduces additional overhead, its effect remains moderate relative to that of the input scale. Moreover, even for the largest evaluated configuration, with $n = 31$ and $N = 2048$, the client-side communication cost is only approximately 1.4 MB. Therefore, the communication overhead of DumboMix remains practical even for mobile-class clients.

Table 5: Total communication (KB) under varying input scales (N) and numbers of servers (n).

# n \ # N	**64**	**128**	**256**	**512**	**1024**	**2048**
4	33.36	66.36	132.36	264.36	528.36	1056.36
7	39.34	77.84	154.84	308.84	616.84	1232.84
10	42.80	84.10	166.60	331.60	661.60	1321.60
16	46.56	90.56	178.56	354.56	706.56	1410.56
22	49.50	94.82	185.68	367.18	730.18	1456.18
31	52.39	98.89	191.89	377.89	749.89	1493.89