$\widetilde{\text{O}}$ptimal Adaptively Secure Hash-based MVBA and Asynchronous Common Subset*

Hanwen Feng[1], Zhenliang Lu[2], and Qiang Tang[3]

[1] University of Canterbury, Christchurch, New Zealand
hanwen.feng@canterbury.ac.nz
[2] City University of Hong Kong, Hong Kong, China
zhenliang.lu@cityu.edu.hk
[3] The University of Sydney, Sydney, Australia
qiang.tang@sydney.edu.au

Abstract. Asynchronous multiparty computation (AMPC) requires an input agreement phase where all participants have a consistent view of the set of private inputs. While the input agreement problem can be precisely addressed by a Byzantine fault-tolerant consensus known as Asynchronous Common Subset (ACS), existing ACS constructions with potential post-quantum security have a large $\widetilde{\mathcal{O}}(n^3)$ communication complexity for a network of n nodes. This poses a bottleneck for AMPC in the same setting. In contrast, ACS has optimal constructions with quadratic communication complexity based on bilinear map assumptions.

In this paper, we bridge this gap by introducing a nearly optimal ACS, which, assuming a common coin oracle, relies on the blackbox use of collision-resistant hash functions. It exhibits $\widetilde{\mathcal{O}}(n^2)$ communication complexity, expected constant round complexity, and security against adaptive adversaries who can corrupt up to $n/3$ nodes and perform "after-the-fact-removal" attacks.

At the core of our new ACS is the first nearly optimal hash-based asynchronous Multi-valued Validated Byzantine Agreement (MVBA). To reduce cubic communication while avoiding heavy cryptographic tools, we introduce a new design paradigm with several novel components. This paradigm also enables an information-theoretic MVBA construction, whose communication complexity is higher than its hash-based counterpart but still quadratic. We give standard treatment of those using the conventional property based definitions. We also define and analyze our MVBA and components within the UC-framework, facilitating their modular use in broader applications, particularly in AMPC.

1 Introduction

Input agreement challenge in asynchronous MPC. In secure multiparty computation (MPC), it is essential for all parties to reach an agreement on the set of "privately committed" inputs first [1], over which the subsequent computation should be carried out. For instance, in MPC based on (threshold) fully homomorphic encryption [50,57], participants must agree on the same set of ciphertexts at the outset, while subsequent computation can be performed locally. Similar challenges exist in secret-sharing-based MPCs [5], where the parties need to decide on the set of secretly shared values. Under the synchronous network assumption, existing MPC protocols handle the input agreement issue by leveraging *broadcast channels*. On the other hand, since the broadcast channel is unavailable in the asynchronous network where messages may be arbitrarily delayed but eventually delivered, asynchronous MPC protocols [8,28] (AMPC) have to apply dedicated asynchronous Byzantine fault-tolerant (BFT) to facilitate the process, which, however, may incur significant overhead and sometimes even dominate the overall communication cost of AMPC.

The core asynchronous BFT protocols employed in all existing AMPC protocols [8,24,28,51] include *Reliable Broadcast* (RBC) [15] and *Agreement on a Core Set* (ACoreSet) [1,8]. RBC can be seen as the asynchronous analog of the broadcast channel, but cannot guarantee termination in the face of a malicious sender. To patch the non-termination issue, the n participants of the AMPC, after disseminating their

* Work done while Hanwen and Zhenliang were both at The University of Sydney.

inputs (e.g., homomorphic ciphertexts, or shares of private inputs) via n RBC instances, will jointly execute an ACoreSet to decide which $n-f$ RBC instances have terminated. Here f is the maximum number of nodes an adversary can corrupt, and we consider the optimal resilience of $f = \lceil \frac{n}{3} \rceil - 1$ throughout this paper. However, running n instances of RBC [4], and ACoreSet used in existing works will each incur $\mathcal{O}(n^3)$-bits communication cost, which is prohibitive when the AMPC is deployed on a moderate scale.

While much MPC research focuses on reducing communication, they are mostly on the parts relevant to the computation task, (while ignoring the cost of the input agreement phase first). However, the actual dominating complexity remains in the input agreement phase. For example, evaluating a circuit with M multiplication gates and D layers can only incur $\mathcal{O}(M \cdot n + D \cdot n^2)$ communication complexity [5], as shown in many AMPC protocols [25–28]. For computational tasks of interest, such as oblivious message permutation for anonymous communication [51], M is merely $\widetilde{\mathcal{O}}(n)$ and D is $O(\log^2 n)$, making the cubic communication cost resulting from input agreement a bottleneck.

Asynchronous common subset. The input agreement problem that arises in the literature of AMPC indeed has broader connections. A BFT consensus primitive, *Asynchronous Common Subset* (ACS), which allows a network of n nodes, each with input, to eventually agree on a set of $n-f$ input values, solves exactly the input agreement problem (and can replace the paradigm of using n RBC instances plus an ACoreSet altogether). Historically, Ben-Or et al. [9] directly used ACS to denote agreement on a core set. Recent asynchronous consensus studies [36, 44, 45, 53, 65] distilled the notion into its current form, and demonstrated it as a core tool for realizing a public distributed ledger in asynchronous networks. Indeed, several ACS constructions [53,65] directly follow similar approaches of using n RBC plus an ACoreSet, which still suffers from the high communication and round costs. However, better ACS constructions beyond this traditional path could, in turn, have implications on the classical input agreement problem, which has not been well explored in AMPC.

MVBA: the only known way towards optimal ACS. Multi-valued Validated Byzantine Agreement (MVBA), originally proposed by Cachin et al [17], allows n parties to jointly decide one common output, which will be guaranteed valid (according to a predefined *predicate*), when all honest parties input valid values. As hinted implicitly in [17], an MVBA protocol can easily imply an ACS protocol by letting each party multicast its input value, and then they jointly execute an MVBA, each using a *vector* of received values as input. Here, the *predicate* could simply be that the vector contains values from $n-f$ parties (e.g., endorsed by digital signatures.). MVBA was long considered of theoretical interest (since the original MVBA [17] itself has a cubic communication complexity), until recently, a sequence of progress re-establish it as a critical component for practical asynchronous consensus [42–45] and beyond [34, 35, 38, 43]. Notably, exciting recent progress on MVBA itself also significantly reduced the communication complexity: Abraham et al. constructed a protocol called VABA that achieved quadratic communication [3], yet it still left the gap for ACS, as now the input (vector) size itself becomes $\mathcal{O}(n)$ when compiling MVBA to ACS, still resulting a $\mathcal{O}(n^3)$ communication for ACS. Finally, Lu et al [52] expanded the communication complexity into detailed terms, and presented an MVBA with optimal communication complexity of $\mathcal{O}(n\ell + n^2\lambda)$ (called Dumbo-MVBA, and an extension framework), where ℓ is the input size [6]. Incorporating Dumbo-MVBA into Cachin et al.'s framework finally yields the first ACS with quadratic communication complexity of $\mathcal{O}(n^2\ell + n^2\lambda)$; and it is also applicable to AMPC.

[4] In this work, we focus on strong adaptive adversaries as per [2], and $\mathcal{O}(n^2)$-bit communication cost is necessary for an adaptively secure RBC and any other consensus.

[5] Here we do not count the communication cost incurred in the offline phase for generating the Beaver triples, which can be performed via pre-processing.

[6] Due to the FLP impossibility, all asynchronous Byzantine agreement protocols are randomized. This also applies to MVBA, which can be easily shown to imply an asynchronous Byzantine agreement [17]. Throughout this paper, we assume a common coin (which is an orthogonal challenge [38]), and omit its cost when computing complexity metrics. This is the same treatment as in the literature of asynchronous consensus protocols, particularly all hash-based ones [36, 37, 49]. Moreover, in AMPC applications, a common coin can be easily implemented with the help of offline pre-processing, thus having no impact on the communication of online phase.

Striving for post-quantum or information-theoretic security. The complexity reductions in [3,52] do not come for free. They heavily rely on cryptographic tools, particularly non-interactive threshold signature, (e.g., BLS from pairing-based assumptions) to generate succinct proofs to compress communication. This puts several restrictions on applying them to AMPC. (1) Besides the need for private setup, one particular constrain is (in)security against quantum attackers, as we do not have any efficient candidate yet for post-quantum non-interactive threshold signatures[7]; while the traditional approach (with high cubic communication) of using RBC and ACoreSet can be information-theoretically (IT) secure or using collision-resistant hash only. And other parts of AMPC, for example, the secret-sharing based framework [7,25,28], the other parts of the online phase besides input agreement, and the whole offline phase could also be information-theoretically secure. In principle, we may consider *interactive* threshold signatures with post-quantum security, however, this itself is a special form of AMPC that may require the input agreement phase to begin with, thus causing a circular problem. (2) Those algebraic operations can introduce substantial computational overhead. For instance, pairing operations are significantly more costly, being approximately 10^5 times slower than hash computations.

There are recently renewed interests in exploring ACS and MVBA in the setting solely using hash functions [33, 36, 37, 49], and information-theoretic settings. Unfortunately, without the help of threshold signatures, existing hash-based optimal-resilient constructions [33,36] suffer from large *cubic* communication complexity. On the other hand, Feng et al. [37] presented a hash-based and quadratic-communication MVBA at the cost of offering *sub-optimal* resilience [8]. This poses a natural question:

Is it possible to design an optimally resilient, sub-cubic, and adaptively secure MVBA *and* ACS *without public-key cryptography or other heavy machinery?*

1.1 Our Contributions

We answer the above question affirmatively by presenting a near-optimal hash-based MVBA, which allows us to obtain a near-optimal hash-based ACS. We detail our contributions in the following.

Nearly optimal hash-based MVBA, ACS and better AMPC. We present the first nearly optimal MVBA protocols with optimal resilience of $f < n/3$, constant rounds, quadratic message complexity, and $\mathcal{O}(\kappa n\ell + \lambda n^2 \log n)$ communication complexity. Here, κ is the statistical security parameter, and λ is the computational security parameter[9]. In practice, κ, λ are usually small constants independent of the number of nodes n. These protocols solely make use of collision-resistant hash functions and achieve security against *strongly adaptive* attackers. The gap towards the optimal quadratic complexity [10] is at the small factor of $\mathsf{max}\{\log n, \kappa\}$. Our MVBA can easily be adapted to be IT secure, at the cost of increasing the communication to $\mathcal{O}(\ell\kappa n^2)$, which still outperforms all existing IT-secure MVBA and may be of independent interest. Please see Section 6.5 for more discussions. We summarize MVBA constructions in Table 1.

We can then instantiate the general compilation to obtain a hash-based ACS, with optimal resilience of $f < n/3$, constant round complexity, $\mathcal{O}(\kappa n^2)$ message complexity, and $\mathcal{O}(\kappa(n^2\ell + \lambda n^2 \log n))$ communication complexity, by using hash-based signatures like [10]. Constructions of ACS based on n instances of RBC [45, 47, 53, 65] incur at least $\mathcal{O}(n^3)$ message and communication complexity, due to the lower-bound cost of RBC [4]. In Table 2, we compare our ACS with existing optimally resilient ACS schemes, where the cost of common coin generation is consistently discounted for all schemes.

[7] We may apply a post-quantum SNARK to compress post-quantum signatures (e.g., [48]), or obtain a lattice-based non-interactive threshold signature via a universal thresholdizer [14], which, however, are computationally expensive and necessitates other setups.

[8] While [37] assumes $f < n/5$, a concurrent work by Komatovic et al. [49] improves the resilience to $f < n/4$ or $f < (\frac{1}{3} - \epsilon)n$ for a constant $\epsilon > 0$, still sub-optimal.

[9] For 40-bit/80-bit statistical security (comparable to 128-bit/256-bit computational security), κ is 69/137. See Section 6.3 for details.

[10] Since every node will receive $\mathcal{O}(\ell)$ bits, and $\mathcal{O}(n^2)$ communication is necessary for any adaptively secure consensus, a lower bound on MVBA is $\mathcal{O}(\ell n + n^2)$.

Table 1. Comparison of the MVBA protocols

Protocol	Resilience	Communication[†]	Message	Round	Crypto Tool
CKPS-MVBA [17]	$f < n/3$	$\mathcal{O}(\ell n^2 + \lambda n^2 + n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	thld. sig.
VABA [3]	$f < n/3$	$\mathcal{O}(\ell n^2 + \lambda n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	thld. sig.
sMVBA [44]	$f < n/3$	$\mathcal{O}(\ell n^2 + \lambda n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	thld. sig.
Dumbo-MVBA [52]	$f < n/3$	$\mathcal{O}(\ell n + \lambda n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	thld. sig.
DYX+22 [35]	$f < n/3$	$\mathcal{O}(\ell n^2 + \lambda n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(\log n)$	DDH, hash
Trivial hash-MVBA[‡]	$f < n/3$	$\mathcal{O}(\ell n + \lambda n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	hash
Fin-MVBA-1 [36]	$f < n/3$	$\mathcal{O}(\ell n^2 + \lambda n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(1)$	hash
Fin-MVBA-2 [36]	$f < n/3$	$\mathcal{O}(\ell n^2 + n^3 \log n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(1)$	none
FLMT24 [37]	$f < n/5$	$\mathcal{O}\big(\ell n + \lambda n^2 \log n\big)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	hash
Reducer [49]	$f < n/4$	$\mathcal{O}\big(\ell n + \lambda n^2 \log n\big)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	hash
Reducer++ [49]*	$f < (\frac{1}{3} - \epsilon)n$	$\mathcal{O}\big(\gamma_\epsilon(\ell n + \lambda n^2 \log n)\big)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\gamma_\epsilon)$	hash
Our hash-MVBA (Section 6)	$f < n/3$	$\mathcal{O}\big(\kappa(\ell n + \lambda n^2 \log n)\big)$	$\mathcal{O}(\kappa n^2)$	$\mathcal{O}(\log \kappa)$	hash
Our IT-MVBA(Section 6.5)	$f < n/3$	$\mathcal{O}\big(\ell \kappa n^2\big)$	$\mathcal{O}(\kappa n^2)$	$\mathcal{O}(\log \kappa)$	none

[†] λ is the computational security parameter. κ introduces a statistical failure of $2^{\Omega(-\kappa)}$. Following the standard practice in asynchronous consensus literature, we assume a common coin and consistently discount the cost of it for all protocols.

[‡] "Trivial hash-MVBA" denotes a variant of Dumbo-MVBA protocol where the threshold signature is instantiated with a concatenation of hash-based digital signatures.

[*] [49] is a concurrent work. In Reducer++, $\gamma_\epsilon = (\lceil \frac{12}{\epsilon^2} \rceil + \lceil \frac{7}{\epsilon} \rceil)^2$, which is millions when setting $\epsilon = \frac{1}{12}$ such that $1/3 - \epsilon = 1/4$. In contrast, to ensure a statistical failure probability of at most 2^{-40} in our protocols, κ only needs to be 69 (see Lemma 8).

We then demonstrate how to use ACS to solve the input agreement problem in secret-sharing-based AMPC. Combining this with existing techniques on function evaluation [7, 32], we obtain the first hash-based online-phase AMPC protocol with a quadratic communication of $\mathcal{O}\big(M \cdot n + \kappa(\ell \cdot n^2 + \lambda n^2 \log n)\big)$, while the previous best one has to include an extra cubic term $\mathcal{O}\big(n^3\big)$ due to the input agreement. Here, M is the number of multiplicative gates in the circuit C, n is the number of nodes, and ℓ is the input size.

In the context of online/offline AMPC, the common coins required by our ACS can be easily implemented using the built-in pre-processing phase. Specifically, all participants can obtain sufficient Beaver triples and pre-shared random values during the offline pre-processing phase (for example, by using protocols such as those in [28]). Then, during the online phase, they can simply multicast their shares and reconstruct a random value that serves as a coin. This approach incurs only an $O(n^2)$ communication cost in the online phase and does not rely on algebraic operations.

Beyond the asymptotic improvements shown in Tables 1 and 2, our designs offer concretely smaller communication costs starting from moderate network scales. For example, according to our concrete analysis, our MVBA surpasses FIN-MVBA in communication efficiency for network sizes as small as $n \approx 105$ under 128-bit security. Please see Section 8 for more details.

A new framework, new components, and UC security. Our advancements in MVBA, to be detailed in Section 1.2, stem from a novel framework significantly divergent from the ones in existing MVBA protocols, and multiple new tools we propose. As we briefly mentioned above, translating the design methodology of existing optimal MVBA in [52] directly into the hash-based setting would result in cubic communication complexity. We present the construction in a modular way, and illustrate how to construct a nearly optimal MVBA using three components: *somewhat-good multi-dealer information dispersal* (SMID, see Section 4), *synchronized multi-valued broadcast* (SMB, see Section 5), and *asynchronous reliable consensus* (ARC, see Appendix E), along with established components like asynchronous binary agreement (ABA) [55] and common coins.

Table 2. Comparison of ACS protocols

Protocol	Resilience	Communication	Message	Round	Crypto Tool
CKPS-ACS [17]	$f < n/3$	$\mathcal{O}(\ell n^2 + \lambda n^3)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	thld. sig.
Dumbo2 [45]	$f < n/3$	$\mathcal{O}(\ell n^2 + \lambda n^3 \log n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(1)$	thld. sig.
sDumbo [44]	$f < n/3$	$\mathcal{O}(\ell n^2 + \lambda n^3 \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	thld. sig.
CKPS-ACS [17] with Dumbo-MVBA [52]	$f < n/3$	$\mathcal{O}(\ell n^2 + \lambda n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	thld. sig.
HBBFT [53]	$f < n/3$	$\mathcal{O}(\ell n^2 + \lambda n^3 \log n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(\log n)$	hash
{HBBFT [53] or PACE [65]} with RBC from [4]	$f < n/3$	$\mathcal{O}(\ell n^2 + n^3 \log n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(\log n)$	none
Fin-ACS [36]	$f < n/3$	$\mathcal{O}(\ell n^2 + \lambda n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(1)$	hash
Fin-ACS [36] with RBC from [4]	$f < n/3$	$\mathcal{O}(\ell n^2 + n^3 \log n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(1)$	none
DDL+24 [33]†	$f < n/3$	$\mathcal{O}(\ell n^2 + \lambda n^3)$	$\mathcal{O}(n^3)$	$\mathcal{O}(1)$	hash
DAG-Rider [47]‡	$f < n/3$	$\mathcal{O}(\ell n^2 + \lambda n^3 \log n)$	$\mathcal{O}(n^3)$	$\mathcal{O}(1)$	hash
Our ACS	$f < n/3$	$\mathcal{O}\big(\kappa(\ell n^2 + \lambda n^2 \log n)\big)$	$\mathcal{O}(\kappa n^2)$	$\mathcal{O}(\log \kappa)$	hash

† Das et al. (DDL+24) can be interpreted as a hash-based ACS protocol (with communication complexity of $\mathcal{O}\big(\ell n^2 + \lambda n^3\big)$) and a hash-based (weak) coin protocol (with communication complexity of $\mathcal{O}\big(\lambda n^3\big)$), while other hash-based or IT protocols focus on the "ACS" part and assume a coin oracle.
‡ DAG-Rider is a hash-based atomic broadcast protocol. Although honest nodes may obtain different outputs at individual decisions, the final ledgers satisfy the mutual prefix property. Each output corresponds to an ACS instance.

To facilitate their modular uses in future applications, particularly in AMPC, we also formulate all these concepts as ideal functionalities within the Universal Composability (UC) framework [21], in contrast with all existing treatments in the literature that are property based. Because of asynchronous communication and adaptive corruption, and inherent limitations of several components, describing those ideal functionalities precisely requires care. We demonstrate that all constructions presented in this paper UC-realize the corresponding functionalities, enabling the protocols to be flexibly composed in larger protocol designs. We believe those new components as well as the simulation-based formulations are of independent interests and could be more broadly useful. For readability, we also provide the conventional treatment using property-based definitions, while deferring all details of the ideal functionalities and UC proofs to Appendices B, C, D, F, and G.

1.2 Challenges and Technical Overview

Conventional design of MVBA. [11] Recall the objective of MVBA, which is to reach consensus and terminate on a "valid" input (when all honest nodes contribute valid inputs). One intuitive approach is to randomly select a node $\mathcal{P}_i$ to provide its input, and let all nodes converge on $\mathcal{P}_i$'s input (e.g., letting $\mathcal{P}_i$ to reliably broadcast his value). This way, there is a constant probability that the selected value is valid, which may be amplified easily, say sample κ nodes to make sure at least one of them is honest and will respond. However, this approach will fail in the face of adaptive adversaries which can corrupt *all* κ (as usually κ is a small constant that could be significantly smaller than $n/3$) selected nodes, prevent them from providing valid inputs, or simply all mute. To counter the adaptive corruption, existing MVBA protocols [3, 17, 36, 44, 52] let all nodes "provably disseminate" their inputs to the network before the election, such that (1) a disseminated input value from an honest node is always available (even if corrupted later after the election), as it can be consistently reconstructed by all others; (2) the "proof" of dissemination also needs to be delivered to all nodes, which will later inform the nodes whether to repeat or not after a random selection.

[11] It is well-known that in the asynchronous setting, constructing a communication efficient MVBA is significantly more complex, than the binary version (e.g., the seminal work of [?, 55, 56]), even with public key cryptographic tools. Let alone using hash only. We give more detailed comparisons/discussions in Section 2.

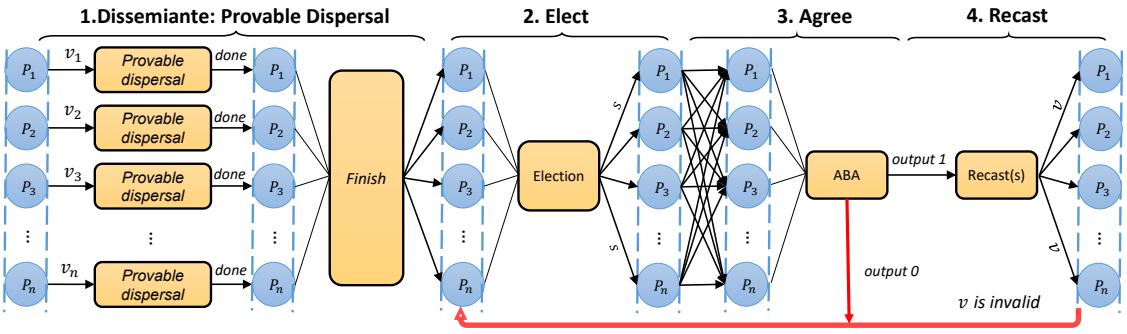

Fig. 1. The execution flow of Dumbo-MVBA [52].

Doing such procedures in a *communication efficient* way requires special care. Particularly, the"optimal" disseminate-and-recast of Dumbo-MVBA [52] is achieved delicately by *provable dispersal*, which makes careful uses of the threshold signature, vector commitment (VC) [22], and the erasure code [11]. Each node shall encode its input into n smaller fragments via an erasure code scheme, and use VC to commit all fragments into a short vc while generating a validity proof for each fragment. Then, each node sends its vc, along with a fragment and the validity proof, to each other node. When receiving a valid fragment from another sender, each node will respond with a partial signature on vc, so each honest node will be able to obtain a threshold signature on his vc as proof to show his dispersal is completed. See Figure 1 as a pictorial illustration.

Challenges of using hash alone to reduce communication. Note that threshold signature plays a vital role in constructing communication efficient MVBA for multiple reasons, including: (1) forming a *proof* to show the validity of each fragment; (2) generating a *proof* to attest the availability of a dispersed value; (3) providing a *certificate* that nodes know how to proceed after selection. Numerically, as shown in Table 1, those threshold-signature based "proofs" are critical to reduce communication in two major parts: the overall communication (e.g., the $O(\lambda n^2)$ term in Dumbo-MVBA) when input size is small; and for extension (e.g., the $O(n\ell)$ term for size ℓ input in Dumbo-MVBA). In the hash-based setting, one might consider using a concatenation of $n-f$ hash-based signatures instead of a threshold signature, however, it will immediately lead to cubic communication.

Now without threshold signatures, we must deviate from the traditional approach to attain the desired communication complexity. Instead, we start with a "weak" dissemination (without a strong proof), but augment the latter phase to be "strong" to still enable nodes to gradually figure out how to proceed after selection, with only some minimal and unreliable hint. Specifically, as outlined in Figure 2, we begin with a "somewhat-good" information dispersal (SMID) as our dissemination phase. SMID replaces partial signatures with simple *echo* messages. However, it does not serve as proof that can be publicly verified, thus cannot even inform the nodes whether a selected node actually completed the dispersal, let alone ensuring there is only one consistent value if jointly reconstructed. The strengthening of the "Agree" part turns out to be highly non-trivial.

Further challenges of adaptive security with sub-cubic communication. Adaptive corruption makes it even more challenging. In the election phase, even if we select a "so-far-honest" node $\mathcal{P}_s$ that honestly sent fragments to $n-f$ nodes, once the adaptive adversary learns that $\mathcal{P}_s$ has been selected (when there could still f honest nodes that have not received anything yet), it can immediately corrupt $\mathcal{P}_s$, and modify the remaining f fragments yet to be delivered. It follows that the remaining f honest nodes will receive fragments corresponding to different values. Now, during recast, it is possible that $f+1$ honest nodes recast and obtain original v_s, while another f honest nodes may recast different values. [12]

To enable nodes to converge to the same output value (for agreement & termination), we proceed in multiple steps: We first introduce a new primitive called SMB (Synchronized Multi-valued Broadcast) and give an efficient construction. An SMB ensures that, if $n-2f$ honest nodes have the same input v, then all

[12] We remark that dealing with adaptive corruption with cubic communication could be easy, as we can simply let each node *reliably broadcast* its full value (without erasure encoding) at the beginning.

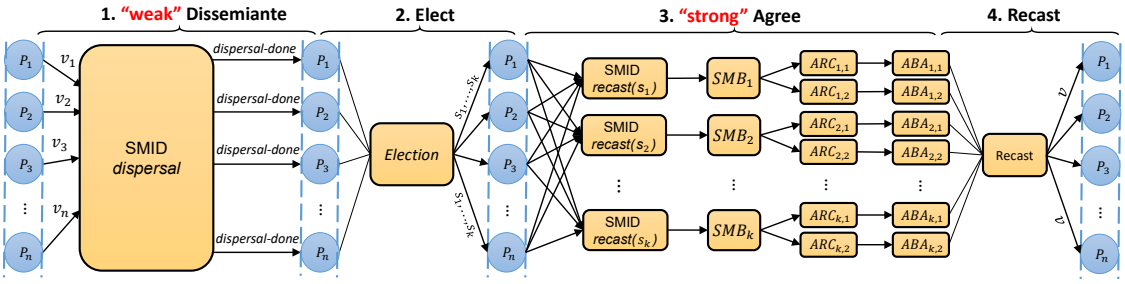

Fig. 2. The execution flow of our MVBA.

honest nodes can output a set with size at most 2. And, for any two honest nodes $\mathcal{P}_i$ and $\mathcal{P}_j$, if $\mathcal{P}_i$ outputs a set $val_i = \{v'\}$, then v' must be subset of $\mathcal{P}_j$'s output set val_j. Additionally, if $|val_i| = |val_j| = 2$, then $val_i = val_j$. The SMB acts as a robust *filtering* mechanism, limiting the input values for the next step among all honest nodes to be at most two. For details about SMB, please see Sec.5. Next, to select one of these values to recast the final output, we make use of and construct an asynchronous reliable consensus (ARC). This is like a consensus analog of RBC that ensures agreement, and a conditional termination, if all honest nodes have the same input value v, they terminate and output v. Considering there are at most two values after SMB, hence, we let all honest nodes participate in two ARC protocols.

Now note that, there is a possibility that one of the two values is not held by all honest nodes, introducing a further termination issue. To help with the further decision, we employ an ABA (Asynchronous Binary Agreement) protocol after each ARC to "vote". The input for $\mathcal{P}_i$ of each ABA instance is 1 if $\mathcal{P}_i$ already outputs in the corresponding ARC. When $\mathcal{P}_i$ outputs 1 in one ABA instance, he will assign 0 as inputs for all remaining ABA instances, if their corresponding ARC instances have not terminated. The discussion above assumes a "so-far-honest" node having completed its dispersal before the election phase and being elected. To ensure this, we adjust the election phase to choose κ random nodes, ensuring that at least one of them to be honest, where κ is a statistical security parameter (usually a small constant). Now since at least one node is "so-far-honest" thus for one ARC instance, sufficient $(n - 2f)$ honest nodes will have the same vc as input, thus the corresponding ARC and ABA will terminate, which in turn helps other remaining instances to terminate.

Further optimization. The above approach uses 2κ ABA instances to check the status of 2κ ARC instances, so that the group can select one ARC with a valid output among them. This process resembles a linear search, and the 2κ ABA instances require $O(\kappa)$ common coins in expectation. We further introduce a binary-search-like strategy that requires only $O(\log \kappa)$ ABA instances; see Section 6.4.

2 Related Work

Byzantine Agreement (BA) [60] is a fundamental problem in Byzantine Fault-Tolerant (BFT) distributed computing. The primary objective of BA is to enable all nodes in a distributed system to reach consensus on their initial input values, even when there is a malicious adversary with control over up to f nodes. This problem has been extensively studied by researchers, leading to investigations in various network models and adversary assumptions. One of the important variants of the BA problem is the Asynchronous Common Subset (ACS) [9]. It plays a crucial role in solving asynchronous Multi-Party Computation (MPC) problems. All existing asynchronous MPC protocols [6, 9, 13, 20, 23, 25–28, 51] utilize ACS to achieve consensus on which private inputs can be used as input for a computation circuit [59].

The earliest ACS protocol relied on n concurrent ABA sub-protocols, resulting in high communication complexity and sub-linear time complexity. Despite the high cost, HBBFT [53] demonstrated certain advantages compared to ACS constructions that reduce to MVBA [17]. The most recent work, Dumbo [45], introduced an improved approach to applying MVBA for constructing ACS, achieving better performance both theoretically and practically. Due to the advantageous features of MVBA, it has become a versatile

and widely applicable primitive in various cryptographic protocols, serving as an essential underlying component, including State Machine Replication (SMR) or Atomic Broadcast (ABC) [18, 42, 44, 53], Distributed Key Generation [34, 35, 43], and Dynamic-committee Proactive Secret Sharing (DPSS) [46, 64].

As a powerful underlying tool, the performance of MVBA significantly influences the efficiency of applications relying on it. Initially proposed in [17] with a communication complexity of $\mathcal{O}(n^2\ell + n^2\lambda + n^3)$, substantial progress has been made to enhance its performance. Initially, Abraham et al. [3] removed a $\mathcal{O}(n^3)$-term, achieving quadratic communication complexity for the first time at $\mathcal{O}(\ell n^2 + \lambda n^2)$. Later, Lu et al. [52] further reduced the $\mathcal{O}(\ell n^2)$-term to $\mathcal{O}(\ell n)$ for optimal communication complexity when $\ell \geq \lambda n$. The MVBA work in [52] marked a milestone for achieving the first optimal ACS with quadratic communication complexity, leveraging the approach presented in [17]. This enabled the realization of ACS with $\mathcal{O}(n^2)$ communication complexity, benefiting from the improved performance of MVBA. Additionally, Guo et al. [44] presented an MVBA with significantly fewer rounds.

However, all these MVBA efforts rely on threshold signatures, which are vulnerable to quantum attacks and necessitate a trusted setup, posing challenges in various applications, such as Distributed Key Generation (DKG).

Several recent works [1, 35, 36] have shifted focus to studying MVBA in the *information-theoretic* (a.k.a. signature-free) setting, or in the *hash-based* setting, where the hash function is the only cryptographic tool and is treated as a black box. These works provide a general framework applicable to both settings, with the hash-based instantiations offering better performance. In particular, Das et al. [35] present MVBA schemes as a component of their distributed key generation protocols, whose hash-based instantiation achieves $\mathcal{O}(\log n)$ rounds, $\mathcal{O}(n^3)$ messages, and communication complexity $\mathcal{O}(\ell n^2 + \lambda n^3)$. Duan et al. [36] further improve this to $\mathcal{O}(1)$ rounds. Nonetheless, a significant asymptotic efficiency gap remains between these hash-based MVBA constructions and classical approaches such as [3, 17, 52].

Duan et al. [36] also proposed two ACS protocols in the IT setting and the hash-only setting, respectively, both derived from their MVBA. The IT-secure version has bit complexity $\mathcal{O}(\ell n^2 + \lambda n^2 + n^3 \log n)$, while the hash-based version achieves $\mathcal{O}(\ell n^2 + \lambda n^3)$ communication complexity. Other hash-based constructions have focused directly on ACS. For example, Zhang et al. [65] gave an ACS with $\mathcal{O}(\log n)$ rounds in the same setting, by running n parallel instances of ABA. Sui and Duan [63] presented ABC protocols with both IT-secure and hash-based instantiations, featuring $\mathcal{O}(n^2)$ message complexity; however, their communication complexity remains $\mathcal{O}(\ell n^2 + \lambda n^3)$.

Very recently, Das et al. [33] presented a hash-based ACS with communication complexity $\mathcal{O}\ell n^2 + \lambda n^3$. Unlike previous works, their protocol does not assume external common coins; instead, they build a weak common coin from scratch as a part of their ACS protocol. Thus, their construction consists of two components: (i) a hash-based weak common coin protocol with complexity $\mathcal{O}(n^3\lambda)$, (ii) the other part which uses the weak coin to realize ACS. Our work advances (ii) by reducing its communication complexity to near-quadratic. It remains an open problem whether the complexity of the first component the weak common coin can be similarly reduced.

Comparison of MVBA, MBA, and ABA. Byzantine agreement's validity is usually defined in a unanimous manner, *i.e.*, if all honest nodes input the same value, they will agree on this value. For the most basic form of handling binary inputs, asynchronous binary agreement (ABA) [54], this validity also guarantees the agreed value must come from an honest input. Nonetheless, for (Multi-valued) byzantine agreement (MBA) [58] a natural extension to ABA for handling multi-bit inputs, the nodes may agree on an arbitrary value, including the default value $\perp$ when they do not have the same inputs. As it is insufficient for many natural cases, this validity of MBA is often called *weak validity*. On the other hand, the *strong validity*, which ensures that the nodes always agree on the input of an honest node, is known to be out of reach for large input sizes [40].

The notion of weak validity in MBA proves insufficient for many practical scenarios. The ideal concept of validity, termed "strong validity", where nodes always unanimously agree on an honest node's input, is known to be unattainable for large input sizes [40]. In response, Cachin introduced "external validity" [17], ensuring that nodes can reach an agreement on a "valid" value adhering to a predefined predicate function, provided all honest nodes input valid values. This extended form of Byzantine Agreement, known as Multi-

Valued Validated Byzantine Agreement (MVBA), addresses the limitations of MBA. It's essential to note that MVBA and MBA are generally incomparable, given that the output of MVBA may be influenced by the adversary in various input scenarios. However, careful predicate design can mitigate this issue by specifying conditions the output must satisfy.

Challenges from ABA, MBA to MVBA. In general, even in the "classical" setting, efficient MVBA designs are different from either ABA or MBA. As ABA only considers binary input, in an asynchronous network, MVBA is qualitatively harder, ABA is usually used as a component in a blackbox way [17, 36, 44, 45]. Also, MBA, only considering weak validity (i.e., could output sth useless such as $\perp$), is also easier (e.g., running ABA for each bit and then applying some extension tricks); while MVBA considers external validity that must output sth useful to satisfy some predicate function. In the more restricted hash based (or signature-free) setting, the situation cannot be easier.

Specifically, our main novelty is a delicate combination (and formulation) of SMID, SMB, ARC, and ABA, which has never appeared or needed in ABA/MBA. Besides, we also need the more sophisticated ARC+ABA to decide, and proper dispersal-recast.

Comparing with the seminal work of (signature-free) ABA and MBA in [56], in our hash-based MVBA, we rely on a SMB component. In signature-free ABA [56], there is another generalization of SBV, called multi-valued validated all-to-all broadcast (MV-broadcast) [56], which ensures all the desired properties without requiring at least $n - 2f$ honest nodes to have the same input value, essentially functioning as SBV without the filter phase. While [56] introduces the reducing broadcast abstraction (RD-broadcast) to limit broadcast values to a constant, it is unsuitable for our filter phase for two main reasons: (1) it reduces values to at most three, whereas we require at most two; otherwise, we cannot guarantee a useful invocation of ARC; (2) it may return $\perp$, which is not allowed in MVBA. Additionally, MV-broadcast incurs a communication complexity as high as $\mathcal{O}(n^3\ell)$, assuming an input value size of ℓ, whereas our SMB achieves a more efficient quadratic communication complexity.

3 Problem Formulation

3.1 System Model

We consider an *asynchronous* network in which nodes are pairwise connected through *authenticated* channels. We assume there are n designated nodes, denoted by $\{\mathcal{P}_i\}_{i\in\{1,\ldots,n\}}$, where the identities of all participating nodes are publicly known. We focus on optimal resilience, meaning that the total number of corrupted nodes is at most $f = \lfloor\frac{n-1}{3}\rfloor$, where the adversary $\mathcal{A}$ is *adaptive* and can corrupt nodes at any point during the protocol's execution. An adaptive adversary can retract any undelivered messages that were sent by a newly corrupted node. During execution, the nodes that have remained honest up to a given point are referred to as *so-far-honest* nodes, while those that remain honest until the end of execution are called *forever-honest* nodes. For simplicity, we use honest nodes to refer to *forever-honest* nodes. Additionally, we use the terms node and party interchangeably, both referring to a participant in the protocol.

To precisely describe the threat model and support the modular use of our protocols and components, we also define all concepts and analyze the constructions presented in this paper within the UC framework [21]. Specifically, we adopt the formulation language from [29], which accurately captures the adversary's ability to delay messages in an asynchronous network. For a background introduction to UC security, we refer readers to Appendix B.

Setup-wise, a standard PKI setup is needed for our ACS protocols. The system also needs a common random string as the key of a collision-resistant hash function; However, the hash key can be generated by a common coin protocol.

3.2 Design Goals

In this paper, we focus on constructing a communication-efficient Asynchronous Multi-valued Validated Byzantine Agreement (MVBA) [3, 17, 52] and Asynchronous Common Subset (ACS) [9, 53], by solely using collision-resistant hash functions.

Multi-valued Validated Byzantine Agreement (MVBA). MVBA is defined w.r.t. a binary predicate Predicate. When each honest node $\mathcal{P}_i$ provides a valid input v_i, i.e., Predicate$(v_i) = 1$, MVBA ensures all honest nodes output a same valid value. Formally, an MVBA protocol satisfies the following properties with all but negligible probability:

- **Termination.** If every honest node p_i inputs an externally valid value v_i, then every honest node outputs a value.
- **External-validity.** If a honest node outputs v, then Predicate$(v) = 1$.
- **Agreement.** Any two distinct honest nodes always output the same value.

Asynchronous common subset (ACS). It enables n nodes each having an input value to eventually agree on the same *multi-set* of $n - f$ values. Formally, the ACS protocol satisfies the following properties with all but negligible probability:

- **Agreement.** If any two honest nodes output sets S and S', respectively, then their outputs must be same, i.e., $S = S'$.
- **Validity.** If an honest node outputs a set S, then $|S| \geq n - f$, and S contains the input values from at least $n - 2f$ honest nodes.
- **Totality.** If at least $n - f$ honest nodes provide an input, then every honest node must output a set S.

Remark 1 (Quality of MVBA*).* In addition to the three MVBA properties defined above, some existing protocols [36, 52] also ensure *quality*, meaning that, with constant probability, the final output is the input of an honest party. We do not target this property, as it is generally unnecessary for downstream applications, such as the construction of ACS protocols. Furthermore, any ACS protocol trivially yields an MVBA protocol with quality simply by selecting a single entry from the output multiset using public randomness.

Coin-aided Model. As highlighted by the well-known FLP impossibility result [39], randomness is necessary for achieving asynchronous Byzantine agreement. Most asynchronous consensus protocols are mostly designed within the common coin-aided model (including the seminal work of [56]), as even assuming an ideal coin, substantial challenges exist, making asynchronous coin design and coin-aided consensus design two orthogonal challenges. This is particularly the case that all recent hash based protocols [36, 37, 49] except [33] are in this same model.

The common coin has various instantiations [28]. In the online/offline framework, the parties could first generate secret shares of a random value in the offline phase (e.g. by following the preprocessing protocol in [28]), and then they can reconstruct the random value in the online phase by multicasting their shares. The coin generation in the online phase only costs $O(n^2)$ communication and does not rely on cryptography. In the rest of the paper, we follow the standard approach and present our consensus protocols in coin-aided model.

3.3 Preliminaries

Notations. We express our protocols through a series of numbered steps. During the execution of such a protocol, a node is expected to iteratively follow these steps in a sequential order, executing each instruction. Certain instructions may have specific preconditions, and if these conditions are not met, the node will skip the corresponding steps. When we say "**Upon**{Condition}{Instruction}", we mean the instruction should be executed every time the condition is triggered. When we say "**Wait**{Condition}{Instruction}", the instruction is supposed to be only executed once no matter how many times the condition may be triggered. In a protocol Π a message with the format (MsgType, `sid`, ...) indicates that the message is associated with an instance of the protocol Π identified by `sid`. Any message exchanged between two parties follows the format (MsgType, `sid`, ...), where `sid` denotes the identifier associated with the protocol instance, and MsgType indicates the type of message. We denote the computational security parameter and the statistical security parameter by λ and κ, respectively. We say a function F defined over positive integers is negligible

in λ, denoted by $|F(\lambda)| < \mathsf{negl}(\lambda)$, if for any polynomial function P, there exists a positive integer N, such that for any $\lambda > N$, it holds that $|F(\lambda)| < \frac{1}{P(\lambda)}$. We say a probability p is overwhelming, if $\mathsf{p} > 1 - \mathsf{negl}(\lambda)$.

Committee Election (Election): A committee election protocol (Election) [45] is employed to return a randomly sampled common subset containing κ elements, denoted as $\{s_1, \ldots, s_\kappa\}$. Formally, a Election protocol with identifier id satisfies the following properties, except with negligible probability:

- **Termination.** Once $n - 2f$ honest parties activate Election with the identifier id, every honest party that activates Election[id] would output a subset which contains κ elements.
- **Agreement.** The outputs of any two honest nodes are the same.
- **Unpredictability.** Before invoked by $n - 2$ honest node, the probability of the adversary to predict the returned committee is at most $1/\binom{n}{\kappa}$.

Asynchronous binary agreement (ABA). In ABA [**?**, 55, 56] among n nodes, if the honest nodes input a single bit, either 0 or 1, then all honest nodes will output a common bit $b \in \{0, 1\}$, where b corresponds to the input of some honest node. Formally, the protocol satisfies the following properties:

- *Agreement.* The outputs of any two honest nodes are the same.
- *Validity.* The output bit must be the input of at least one honest node.
- *Termination.* All honest nodes will output a bit as long as every honest node provides an input bit.

Asynchronous Reliable Consensus (ARC). In the ARC protocol [12,33], each node takes a value as input and may decide on a common value. Formally, the protocol is designed to satisfy the following properties, except with negligible probability:

- **Totality.** If an honest node outputs v, then all honest nodes output v'.
- **Agreement.** If any two honest nodes output v and v' respectively, then $v = v'$.
- **Validity.** If all honest nodes with the same input value v, then all honest nodes output v.
- **Justification.** If an honest node outputs v, then v is the input of at least $n - 2f$ honest node.

The ARC protocol was first studied and constructed by Blum et al. [12], where the proposed version is computationally secure, randomized, and does not withstand the adaptive adversary considered in this work. To address this limitation, we design a new construction that resists adaptive adversaries while being both information-theoretically secure and deterministic. Our construction is similar to the one-sided voting introduced in [62]; however, it specifically focuses on unitary inputs. The details of our ARC construction[13] are presented in Algorithm 7.

Erasure code. A (k, n)-erasure code [11] comprises two deterministic algorithms, denoted as EC.Encode and EC.Decode. The EC.Encode algorithm takes a data value $\mathbf{m}$ with k data fragments $(m_1, \cdots, m_k)$ as input and outputs n coded fragments $\mathbf{c} = (\mathsf{c}_1, \ldots, \mathsf{c}_n)$. Any k elements in the code vector $\mathbf{c}$ can reconstruct the original data $\mathbf{m}$ using the EC.Decode algorithm. Formally, a (k, n)-erasure code involves two deterministic algorithms:

1. $\mathsf{EC.Encode}(\mathbf{m}, n, k) \to \mathbf{c}$: Given a data $\mathbf{m} \in \mathcal{B}^k$ as input, this deterministic encoding algorithm outputs a vector $\mathbf{c} := \{c_1, \cdots, c_n\}$.
2. $\mathsf{EC.Decode}(\{(i, c_i)\}_{i \in S}, n, k) \to \mathbf{m}$: Given a set $\{(i, c_i)\}_{i \in S}$ where $S \subset [n]$ and $|S| = k$ as input, this deterministic decoding algorithm outputs $\mathbf{m}$.

Correctness. Assuming $\mathcal{B}$ is the field of each fragment, for any $\mathbf{m} \in \mathcal{B}^k$ and any $\mathbb{I} \subset [n]$ with $|\mathbb{I}| = k$, then

$$\Pr[\mathsf{EC.Decode}(\{(i, \mathsf{c}_i)\}_{i \in \mathbb{I}}) = \mathbf{m} \mid \mathbf{c} := (\mathsf{c}_1, \cdots, \mathsf{c}_n) \leftarrow \mathsf{EC.Encode}(\mathbf{m})] = 1.$$

[13] A concurrent work [33] also introduces an ARC that achieves the same goal as our work, while we additionally provide the ARC ideal functionality in Appendix E.

Vector commitment (VC). A VC scheme [22] consists of a tuple of algorithms: (VC.Setup, VCom, Open, VerifyOpen), where VC.Setup produces a public parameter p, VCom produces a vector commitment by committing to any n-sized vectors, Open is used to generate a proof for its specified position value, and VerifyOpen is used to verify that the given value indeed is the specified position value corresponding to the vector commitment. Formally, a VC scheme can be abstracted as a tuple comprising the following algorithms:

1. $\mathsf{VC.Setup}(\lambda, n, \mathcal{M}) \to \mathsf{p}$. Given the security parameter λ, the size n of the input vector, and the message space $\mathcal{M}$ of each vector element, it outputs the public parameter p, which is an implicit input to all other algorithms.
2. $\mathsf{VCom}(\mathbf{m}) \to \mathsf{vc}$. Taking a vector $\mathbf{m} = (m_1, ..., m_n)$ as input, this algorithm produces a commitment vc.
3. $\mathsf{Open}(\mathsf{vc}, m_i, i) \to \pi_i$. Given m_i, position i, and commitment vc, this algorithm generates an opening string π_i serving as proof that m_i is the i-th committed element.
4. $\mathsf{VerifyOpen}(\mathsf{vc}, m_i, i, \pi_i) \to 0/1$. Given m_i, i, commitment vc, and opening proof π_i, it outputs 1 if $\mathsf{Open}(\mathsf{vc}, m_i, i) = \pi_i$, otherwise, it outputs 0.

Correctness. VC is correct, if for all $\mathbf{m} \in \mathcal{M}^n$ and $i \in [n]$,

$$\Pr[\mathsf{VerifyOpen}(\mathsf{vc}, m_i, i, \mathsf{Open}(\mathsf{vc}, m_i, i)) = 1 \mid \mathsf{vc} \leftarrow \mathsf{VCom}(\mathbf{m})] = 1.$$

Position binding. VC is position binding, if for any PPT adversary $\mathcal{A}$,

$$\Pr\left[\begin{array}{c}(\mathsf{vc}, i, m, \pi, m', \pi') \leftarrow \mathcal{A}(1^\lambda) : \mathsf{VerifyOpen}(\mathsf{vc}, m, i, \pi) = 1 \\ \wedge \ m \neq m' \ \wedge \ \mathsf{VerifyOpen}(\mathsf{vc}, m', i, \pi') = 1\end{array}\right] \leq \mathsf{negl}(\lambda).$$

4 Somewhat-good Multi-dealer Information Dispersal

In this section, we introduce our first component: Somewhat-good Multi-dealer Information Dispersal (SMID). Similar to other information dispersal protocols [4, 19, 52], SMID consists of two phases: *dispersal* and *recast*. However, unlike these dispersal protocols that involve a single sender, SMID features n senders, where each participant acts as a sender to encode its input into multiple fragments, and then distributes them across the network.

A straightforward reduction to SMID involves running n asynchronous verifiable information dispersal (AVID) protocols in parallel, with each node acting as the sender for its corresponding dispersal protocol. The classical AVID ensures that once an honest node completes the dispersal phase, all honest nodes can retrieve the same value, regardless of whether the sender is honest. However, dispersing an ℓ-bit value using AVID incurs at least $\mathcal{O}(\ell n + n^2)$ bits of communication cost [4]. When considering n concurrent instances of AVID, results in a cubic communication overhead, contradicting our goal of achieving MVBA with sub-cubic complexity in the hash-based setting.

The cubic communication cost of AVID arises from its strong guarantee of "availability", ensuring that any honest node can always retrieve the same dispersed value, regardless of whether the original sender was honest. However, we relax this guarantee in SMID to achieve a more communication-efficient protocol. In our approach, each honest node acts as a sender, dispersing its input value across the network. However, as a trade-off, the asynchronous setting and adaptive corruption introduce subtle challenges, allowing the adversary to influence what is recast when an honest node attempts to retrieve a value. Hence, in the recast phase, it only ensures that at least $f + 1$ honest nodes can retrieve the same value, rather than enforcing this guarantee for all honest nodes.

To illustrate this, consider the best-effort dispersal process: an honest node $\mathcal{P}_i$ sends fragments of its input to all other nodes. The honest node receives a fragment from the sender and then sends a response back to the sender. $\mathcal{P}_i$ considers the dispersal completed, meaning the fragments have been successfully delivered to the network, once it has received responses from at least $n - f$ distinct nodes. However, among these responses, only $n - 2f$ may come from honest nodes, while up to f honest but "unfortunate" nodes might not receive

the corresponding fragments. When the network later attempts to retrieve the value dispersed by $\mathcal{P}_i$, an adaptive adversary can corrupt $\mathcal{P}_i$ and send modified fragments to these f "unfortunate" nodes, causing them to reconstruct and recast a different value. Formally, the SMID protocol consists of two subprotocols: SMID-dispersal and SMID-recast. The formal definition is as follows:

Definition 1. *A somewhat-good multi-dealer information dispersal (*SMID*) protocol involves n nodes, denoted as $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$, where at most $f < n/3$ nodes are Byzantine. This protocol consists of two subprotocols:* SMID-dispersal *and* SMID-recast*.*

- *In the* SMID-dispersal*, each node $\mathcal{P}_i$ holds an input value v_i and aims to disperse this value to all nodes.*
- *In the* SMID-recast*, any node can take an index i as input, then at least $n - 2f$ honest nodes can recast the value v_i that was dispersed by $\mathcal{P}_i$ if $\mathcal{P}_i$'s dispersal is completed and he has not been corrupted before the recast phase.*

The SMID *protocol (*SMID-dispersal*,* SMID-recast*) is designed to satisfy the following properties, except with negligible probability:*

- ***Termination.*** *All honest nodes will eventually complete the dispersal phase of an* SMID *instance if each honest node $\mathcal{P}_i$ inputs a value v_i.*
- ***Validity.*** *Let $\{\mathcal{P}_i\}_{i\in\mathbb{H}}$ be the set of so-far-honest nodes before the recast phase, and each $\mathcal{P}_i$ has an initial input value v_i for dispersal. If one honest node has completed the dispersal phase, then there exists a subset $\mathbb{I} \subset \mathbb{H}$ with $|\mathbb{I}| \geq f + 1$, such that: if all honest nodes input the same index i in this subset $\mathbb{I}$ during the recast phase, at least $n - 2f$ honest nodes can obtain the same v_i, which is the original input of $P_i, i \in \mathbb{I}$, even if P_i has been corrupted later during the execution.*

4.1 Details of the SMID protocol

We now present SMID. The detailed procedure for SMID can be found in Algorithm 1. The only cryptographic tool in the construction is a vector commitment scheme that helps the network identify the correct fragments sent by an honest sender. The technique is standard and has been employed in many previous designs [52,58]. The details of SMID-dispersal and SMID-recast are as follows:

1. *Fragment phase* (lines 1-5). When an honest node $\mathcal{P}_i$ receives a valid input value v, it initially computes the codewords of value v using EC.Encode. Subsequently, it computes the corresponding vector commitment and position proofs. $\mathcal{P}_i$ then disperses its input value v through FRAGMENT messages, which include a vector commitment, a codeword, and a position proof.
2. *Ok phase* (lines 6-9). For any honest node $\mathcal{P}_i$, upon receiving a valid FRAGMENT message from $\mathcal{P}_j$, it records the content of the message as $fragment[j]$ and subsequently sends an OK back to $\mathcal{P}_j$.
3. *Completed phase* (lines 10-13). For any honest node $\mathcal{P}_i$, upon receiving $n - f$ OK messages from distinct nodes, it multicasts a COMPLETED message.
4. *Output phase* (lines 14-17). For any honest node $\mathcal{P}_i$, upon receiving $n - f$ COMPLETED messages from distinct nodes, it returns (`disperse-done`, id).
5. *Recast phase* (lines 18-26). For any honest node $\mathcal{P}_i$, if it decides to recast $\mathcal{P}_s$'s input, it first checks whether $fragment[s] \neq \perp$. If yes, it multicasts it via a RECAST message and then waits for $f + 1$ valid RECAST messages from distinct nodes. Afterward, it decodes these received codewords and outputs the result. If $fragment[s] = \perp$, then the honest node doesn't produce any output when invoking SMID-recast(id, s).

4.2 Security and Complexity analysis

Theorem 1. *Assuming the underlying hash function is collision resistant, the Algorithm 1 implements a Somewhat-good Multi-dealer Information Dispersal protocol (cf. Definition 1), in the presence of a computationally bounded and adaptive adversary who may corrupt up to $f < n/3$ parties.*

Algorithm 1 The SMID protocol with identify id for $\mathcal{P}_i$

Initialize: $fragment[j] \leftarrow \perp$ for $j \in [n]$

▷ SMID-dispersal[id](v_i)

upon receiving input v_i s.t. $\mathsf{Predicate}(v_i) = true$ **do**
  Let $\boldsymbol{c} := [c_1, c_2, \ldots, c_n] \leftarrow \mathsf{EC.Encode}(v_i, n, f+1)$
  Let $\mathsf{vc} \leftarrow \mathsf{VCom}(\boldsymbol{c})$
  for $j \in [n]$, $\pi_j \leftarrow \mathsf{Open}(\mathsf{vc}, c_j, j)$
  send $(\textsc{Fragment}, \mathsf{vc}, c_j, \pi_j)$ to $\mathcal{P}_j$ for every $j = 1, \ldots, n$
upon receiving $(\textsc{Fragment}, \mathsf{vc}, c_i, \pi_i)$ from $\mathcal{P}_j$ for the first time **do**
  if $\mathsf{VerifyOpen}(\mathsf{vc}, c_i, i, \pi_i) = 1$ **then**
    $fragment[j] \leftarrow (\mathsf{vc}, c_i, \pi_i)$
    send $\textsc{Ok}$ to $\mathcal{P}_j$
upon receiving $\textsc{Ok}$ from $\mathcal{P}_j$ for the first time **do**
  $S_{ok} \leftarrow \mathsf{S}_{ok} \cup \{j\}$
  if $|S_{ok}| = n - f$ **then**
    multicast $\textsc{Completed}$ to all
upon receiving $\textsc{Completed}$ from $\mathcal{P}_j$ for the first time **do**
  $S_{comp} \leftarrow S_{comp} \cup \{j\}$
  if $|S_{comp}| = n - f$ **then**
    return $(\texttt{disperse-done}, \mathsf{id})$

upon receiving input s **do** ▷ SMID-recast[id](s)
  if $fragment[s] := (\mathsf{vc}, c_i, \pi_i)$ **then** ▷ $fragment[s] \neq \perp$
    multicast $(\textsc{Recast}, s, \mathsf{vc}, c_i, \pi_i)$ to all
    upon receiving $(\textsc{Recast}, s, \mathsf{vc}', c_j, \pi_j)$ from $\mathcal{P}_j$ for the first time **do**
      if $\mathsf{vc} = \mathsf{vc}'$ and $\mathsf{VerifyOpen}(\mathsf{vc}, c_j, j, \pi_j) = 1$ **then**
        $Recast_s \leftarrow Recast_s \cup \{(j, c_j)\}$
        upon $|Recast_s| = f + 1$ **do**
          $v_s \leftarrow \mathsf{EC.Decode}(Recast_s, n, f+1)$
          return $(\texttt{recast-output}, s, v_s)$

Proof. We prove each property separately.

1. **Termination.** Since every honest node $\mathcal{P}_i$ has an input value v_i, following the protocol, $\mathcal{P}_i$ multicasts $\textsc{Fragment}$ messages to all. Upon receiving a valid $\textsc{Fragment}$ message from $\mathcal{P}_i$, any honest node $\mathcal{P}_j$ sends an $\textsc{Ok}$ back to $\mathcal{P}_i$. With at most f malicious nodes, $\mathcal{P}_i$ can receive $n-f$ $\textsc{Ok}$ messages from distinct nodes. Afterward, $\mathcal{P}_i$ multicasts a $\textsc{Completed}$ message to all. Again, since the number of malicious nodes is at most f, so at least $n - f$ honest nodes will multicast $\textsc{Completed}$ messages to all. Consequently, all honest nodes eventually receive at least $n - f$ $\textsc{Completed}$ messages from distinct nodes, leading to the completion of the dispersal phase for all honest nodes in an SMID instance.
2. **Validity.** To prove validity, we first show the existence of a subset $\mathbb{I}$ as specified in the definition. If an honest node $\mathcal{P}_i$ has completed the dispersal phase, according to the code, it means that $\mathcal{P}_i$ received $n - f$ $\textsc{Completed}$ messages from distinct nodes. Given that there are at most f malicious nodes, at least $n - 2f$ (forever) honest nodes multicast $\textsc{Completed}$ messages. Let $\mathbb{I}$ denote the set of indices of these $n - 2f$ honest nodes.

   Next, we show that for any $i \in \mathbb{I}$, if all honest nodes input the same index i during the recast phase, then at least $n - 2f$ honest nodes can obtain v_i, and moreover v_i is not controlled by the adversary. Following the procedure outlined in Algorithm 1, when one honest node $\mathcal{P}_i$ multicasts a $\textsc{Completed}$ message, it implies that at least $n - 2f$ honest nodes received the $\textsc{Fragment}$ message from $\mathcal{P}_i$, and these $\textsc{Fragment}$ messages contain the same vector commitment (vc). Since at least $n-2f$ honest nodes have received the fragment with the same vc, which is the vector commitment of value v_i, during the recast phase, every honest node multicasts its $fragment[i]$ whenever $fragment[i] \neq \perp$. Consequently,

since at least $n - 2f$ honest nodes receive valid fragment messages from $\mathcal{P}_i$, each such node can obtain at least $n - 2f > f + 1$ valid fragments that share the same vector commitment vc. This suffices to reconstruct the same value v_i. Therefore, the value v_i is not controlled by the adversary, since vc is the vector commitment to the original value v_i provided by P_i. □

Complexity analysis. In SMID-dispersal, each honest node can send at most $\mathcal{O}(n)$ FRAGMENT messages, $\mathcal{O}(n)$ OK messages, and $\mathcal{O}(n)$ COMPLETED messages. Here, the size of the input value v is ℓ, and the size of a FRAGMENT is $\mathcal{O}(\ell/n + \log n\lambda)$, while both OK and COMPLETED messages have a size of $\mathcal{O}(1)$. Consequently, the message complexity is $\mathcal{O}(n^2)$, and the communication complexity is $\mathcal{O}(n\ell + n^2 \log n\lambda)$. In SMID-recast, for any single input index s, each honest node sends at most $\mathcal{O}(n)$ RECAST messages, where the size of a RECAST is $\mathcal{O}(\ell/n + \log n\lambda)$. As a result, the message complexity is $\mathcal{O}(n^2)$, and the communication complexity is $\mathcal{O}(n\ell + n^2 \log n\lambda)$.

5 Synchronized Multi-valued Broadcast

In this section, we introduce our main new ingredient of MVBA: the Synchronized Multi-valued Broadcast (SMB), which is a generalization of Synchronized Binary-value Broadcast (SBV), a core component of a classic ABA protocol [**?**].

Assume that n nodes have dispersed their inputs through SMID-dispersal. When the network is trying to recast a value, it could result in $f+1$ honest nodes having the correct value while other f honest nodes have arbitrary values provided by the adversary; thus, there could be up to $f + 1$ different values held by honest nodes. Looking ahead, in our MVBA construction, we hope the network will agree on one valid value. SMB is the tool we utilize to "winnow" the values, such that only a few values are left after SMB. At the same time, we hope that, after SMB, the nodes have been "synchronized" about the input values, *i.e.*, if an honest node terminates on a single value val, then this value must appear in any other honest node's output. The formal definition of SMB is as follows:

Definition 2. *A Synchronized Multi-valued Broadcast (*SMB*) protocol involves n nodes, denoted as $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$, where at most $f < n/3$ nodes are Byzantine. Specifically, each node $\mathcal{P}_i$ may be provided with an input v_i (which could initially be empty for some nodes when they participate), and each $\mathcal{P}_i$ will output a set val_i of values. The protocol is designed to satisfy the following properties, except with negligible probability:*

- ***Termination.*** *If at least $n - 2f$ forever honest nodes have the same input value v, then every honest node $\mathcal{P}_i$ will terminate and output a set val_i.*
- ***Obligation.*** *If at least $n-2f$ forever honest nodes have the same input value v, then the set val_i returned by an honest node $\mathcal{P}_i$ is not empty, and $|val_i| \leq 2$.*
- ***Validity.*** *If at least $n-2f$ forever honest nodes have the same input value v, and $\mathcal{P}_i$ and $\mathcal{P}_j$ are honest nodes with $|val_i| \leq |val_j|$, then $val_i \subseteq val_j$. Additionally, if $|val_i| = 2$, then $v \in val_i$.*
- ***Justification.*** *If $\mathcal{P}_i$ is an honest node, then for every $v \in val_i$, v is the input of some honest node.*

Construction Overview. Our construction can be viewed as a generalization of the SBV [**?**], which exhibits the following functionality: each node inputs a binary value and outputs a set of binary values. The main challenge of SMB lies in the *multi-valued* setting, where there could be many different input values, whereas SBV is designed to handle only two possible inputs. To address this, we introduce a "filter" mechanism to reduce the number of values, which works as follows: every node multicasts its input and will echo the values sent by at least $f + 1$ nodes; then, only the values echoed by at least $n - f$ nodes will be processed after the filter. In the case that $f + 1$ of all honest nodes share the same input value, it is easy to argue that there are at most two values after filtering.

Assume v_1 and v_2 are the values left. Now, different honest nodes may have $\{v_1\}$, $\{v_2\}$, or $\{v_1, v_2\}$, while their outputs are supposed to be "inclusive". In other words, we should avoid the case where an honest node outputs $\{v_1\}$ while another honest node outputs $\{v_2\}$. We adopt the techniques from SBV to guarantee the

Algorithm 2 The SMB protocol with identifier id for $\mathcal{P}_i$

$values_i \leftarrow \{\}$, and $auxs_i \leftarrow \{\}$
1: **upon** receiving input v_i **do**
2: **multicast** (FILTER, id, v_i) to all
3: **upon** receiving (FILTER, id, v') from $n-2f$ distinct nodes **do**
4: **multicast** (FILTERECHO, id, v') to all
5: **upon** receiving (FILTERECHO, id, v) from $n-f$ distinct nodes **do**
6: **multicast** (VAL, id, v) to all
7: **upon** receiving (VAL, id, v) from $n-2f$ distinct nodes **do**
8: **if** (VAL, id, v) not yet sent **then**
9: **multicast** (VAL, id, v) to all
10: **upon** receiving (VAL, id, v) from $n-f$ distinct nodes and $v \notin values_i$ **do**
11: $values_i \leftarrow values_i \cup \{v\}$
12: **wait** until $values_i \neq \emptyset$
13: **multicast** (AUX, id, ω), where $\omega \in values_i$
14: **upon** receiving (AUX, id, ω) from $\mathcal{P}_j$ for the first time **do**
15: **if** weight_ω has not been initialized **then**
16: Let $\mathsf{weight}_\omega \leftarrow 0$
17: $auxs_i \leftarrow auxs_i \cup \{\omega\}$, and $\mathsf{weight}_\omega = \mathsf{weight}_\omega + 1$
18: **wait** until $\sum_{\omega \in val_i} \mathsf{weight}_\omega \geq n-f$, for $val_i = values_i \cap auxs_i$
19: **return** val_i

inclusion property, which works as follows: First, each node will maintain a local set of candidate values. Then, each node will endorse exactly one candidate value through an AUX message. Finally, an honest node will output a subset of candidate values endorsed by $n-f$ AUX messages. If an honest node outputs a single value v_1, there must be at least $n-f$ AUX carrying v_1. Consequently, every other node must include v_1 in its output, ensuring its inclusion. It is worth noting that our *validity* property inherently guarantees this inclusion.

Furthermore, we note the following: For any honest node $\mathcal{P}_i$, as proven in Lemma 1, each honest node can send at most one FILTER message, two FILTERECHO messages, three VAL messages, and one AUX message. Consequently, if an honest node $\mathcal{P}_i$ has already received one FILTER message from $\mathcal{P}_j$, it will disregard any additional FILTER messages from $\mathcal{P}_j$. Similarly, for other message types, $\mathcal{P}_i$ will only consider at most the maximum allowable number of messages received from other nodes. Our protocol implicitly follows this rule.

5.1 Details of the SMB protocol

According to the definition of SMB, except for the Justification property, all other properties are satisfied under the condition that at least $n-2f$ honest nodes have the same input value. Although it is a weaker version of MV-broadcast, it is sufficient for us to build our MVBA in Section 6. In this section, we provide a construction for SMB with IT-security, and the detailed procedure for SMB can be found in Algorithm 2. Here is a comprehensive description:

1. *Filter phase* (lines 1-2). All honest nodes multicast their input via FILTER messages.
2. *FilterEcho phase* (lines 3-4). For any honest node $\mathcal{P}_i$, if it receives $n-2f$ FILTER messages carrying the same value v' from distinct nodes, then it will multicast the value v' via a FILTERECHO message.
3. *Val phase* (lines 5-9). For any honest node $\mathcal{P}_i$, if it receives $n-f$ FILTERECHO messages carrying the same value v from distinct nodes, then it will multicast the value v via a VAL message. Besides, upon receiving $n-2f$ VAL messages carrying the same value v from distinct nodes and not having multicast (VAL, id, v), it will multicast a VAL message along with the value v.
4. *Aux phase* (lines 10-13). For any honest node $\mathcal{P}_i$, if it receives $n-f$ VAL messages carrying the same value v from distinct nodes, then it will add the value v to the $value_i$ set. If the $value_i$ set is not empty, it will multicast (AUX, id, ω) to all, where $\omega \in values_i$.

5. *Output phase* (lines 14-19). For any honest node $\mathcal{P}_i$, it counts the number weight_w of AUX messages carrying the value w, and records all received w in the set aux_i. $\mathcal{P}_i$ outputs the intersection val_i of aux_i and $values_i$, only when it has received at least $n-f$ AUX messages that carry values already in $values_i$, i.e., $\sum_{w \in val_i} \mathsf{weight}_w >= n-f$.

5.2 Security and Complexity analysis

Lemma 1. *In the worst case, each honest node sends at most one* FILTER *message, two* FILTERECHO *messages, three* VAL *messages, and one* AUX *message.*

Proof. Each honest node first sends a single FILTER message, as every honest node has exactly one input. Second, it is important to emphasize that each honest node $\mathcal{P}_i$ can multicast at most two FILTERECHO messages. This limitation arises because every honest node receives a total of n FILTER messages, among which at most two distinct values may appear at least $n-2f$ times.

Furthermore, in the worst case, since each honest node $\mathcal{P}_i$ can multicast at most two FILTERECHO messages, the total number of FILTERECHO messages from honest nodes is at most $2(n-f)$. According to the protocol, an honest node sends a VAL message if either (i) it receives at least $n-f$ FILTERECHO messages carrying the same value from distinct nodes, or (ii) it receives VAL messages with the same value from at least $n-2f$ distinct nodes. Since at most f nodes are malicious, this implies that an honest node sends a VAL message if either (i) it receives at least $n-2f$ FILTERECHO messages with the same value from honest nodes, or (ii) it receives a VAL message with that value from at least one honest node. Thus, in any scenario, once an honest node sends a VAL message, at least $n-2f$ honest nodes must have already sent FILTERECHO messages with the same value. As a result, each honest node can send at most $\frac{2(n-f)}{n-2f}$ VAL messages. Given that we assume $n \geq 3f+1$, this implies that every honest node can send at most three VAL messages. Finally, by the code, each honest node sends only one AUX message. □

Lemma 2. *If at least $n-2f$ honest nodes have the same input value v, then every honest node $\mathcal{P}_i$ can multicast at most two different* VAL *messages to all, where the* $(\text{VAL}, \mathsf{id}, v)$ *message is eventually mulitcast. Additionally, if one honest node $\mathcal{P}_i$ multicasts* $(\text{VAL}, \mathsf{id}, v')$ *to all and $v \neq v'$, and another honest node $\mathcal{P}_j$ multicasts* $(\text{VAL}, \mathsf{id}, v'')$ *to all with $v \neq v''$, then $v' = v''$.*

Proof. If at least $n-2f$ honest nodes have the same input value v, then every honest node can receive $n-2f$ $(\text{FILTER}, \mathsf{id}, v)$ messages from distinct nodes, resulting in every honest node multicasting $(\text{FILTERECHO}, \mathsf{id}, v)$ to all. Due to $n \geq 3f+1$, according to the code, every honest node can multicast at most two different FILTERECHO messages. Since every honest node multicasts $(\text{FILTERECHO}, \mathsf{id}, v)$, every honest node can receive $n-f$ $(\text{FILTERECHO}, \mathsf{id}, v)$ messages from distinct nodes; hence, every honest node will multicast a $(\text{VAL}, \mathsf{id}, v)$ message to all.

Once again, if one honest node $\mathcal{P}_i$ multicasts $(\text{VAL}, \mathsf{id}, v')$ to all and $v \neq v'$, and another honest node $\mathcal{P}_j$ multicasts $(\text{VAL}, \mathsf{id}, v'')$ to all with $v \neq v''$, then it implies that $\mathcal{P}_i$ received $n-f$ $(\text{FILTERECHO}, \mathsf{id}, v')$ from distinct nodes, and $\mathcal{P}_j$ received $n-f$ $(\text{FILTERECHO}, \mathsf{id}, v'')$ from distinct nodes. According to the previous analysis, every honest node can multicast at most two FILTERECHO messages, and one of them must be $(\text{FILTERECHO}, \mathsf{id}, v)$. As a result, at least one honest node will multicast $(\text{FILTERECHO}, \mathsf{id}, v)$, $(\text{FILTERECHO}, \mathsf{id}, v')$, and $(\text{FILTERECHO}, \mathsf{id}, v'')$. This leads to a contradiction unless $v' = v''$. □

Lemma 3. *If one honest node multicasts* $(\text{AUX}, \mathsf{id}, v)$ *to all, then for any honest node $\mathcal{P}_i$, it will eventually be the case that $v \in values_i$.*

Proof. If an honest node multicasts $(\text{AUX}, \mathsf{id}, v)$ to all, according to the code, this honest node received $n-f$ $(\text{VAL}, \mathsf{id}, v)$ messages from distinct nodes. This implies that at least $n-2f$ honest nodes multicast $(\text{VAL}, \mathsf{id}, v)$ to all honest nodes. Subsequently, all honest nodes will multicast $(\text{VAL}, \mathsf{id}, v)$, following the procedure outlined in Algorithm 2. It also indicates that all honest nodes can receive $(\text{VAL}, \mathsf{id}, v)$ from $n-f$ honest nodes. Consequently, v will be added to the set *values*. □

Lemma 4. Justification. *If $\mathcal{P}_i$ is an honest node, then for every $v \in val_i$, v is the input of some honest node.*

Proof. If an honest node $\mathcal{P}_i$ outputs val_i, according to the code, $val_i \subseteq values_i$. For any element $v \in values_i$, it implies that $\mathcal{P}_i$ received $n-f$ $(\text{VAL}, \mathsf{id}, v)$ messages from distinct nodes. According to Algorithm 2, at least one honest node $\mathcal{P}_j$ received $n-f$ $(\text{FILTERECHO}, \mathsf{id}, v)$ messages from distinct nodes. This implies that at least one honest node multicasted a $(\text{FILTERECHO}, \mathsf{id}, v)$ message, signifying that this honest node received $n-2f$ $(\text{FILTER}, \mathsf{id}, v)$ messages from distinct nodes. Since there are at most f malicious nodes, it follows that at least one honest node's input is v. □

Lemma 5. Termination. *If at least $n-2f$ honest nodes have the same input value v, then every honest node $\mathcal{P}_i$ will terminate and output a set val_i.*

Proof. If at least $n-2f$ honest nodes have the same input value v, then following the procedure outlined in Lemma 2, all honest nodes eventually multicast $(\text{VAL}, \mathsf{id}, v)$ message, resulting in all honest nodes eventually adding v to the set *values*. However, it is possible that some honest nodes multicast $(\text{VAL}, \mathsf{id}, v')$ messages, where $v \neq v'$. Consequently, it is also possible that $v' \in values_i$. Therefore, all honest nodes will multicast a AUX message. Now consider two cases:

- **Case 1:** If some honest nodes multicast $(\text{AUX}, \mathsf{id}, v)$ messages, while another set of honest nodes multicast $(\text{AUX}, \mathsf{id}, v')$ messages, following the procedure outlined in Lemma 3, then for any honest node $\mathcal{P}_i$, the set $values_i$ will eventually contain both v and v'.
- **Case 2:** If all honest nodes can multicast $(\text{AUX}, \mathsf{id}, \bar{v})$ messages, where $\bar{v} = v$ or v', it is clear that all honest nodes will have $values_i = \{\bar{v}\}$.

In either case, every honest node $\mathcal{P}_i$ will satisfy the condition in line 18 of Algorithm 2, ensuring that every honest node $\mathcal{P}_i$ will terminate and output a set val_i. □

Lemma 6. Obligation. *If at least $n-2f$ honest nodes have the same input value v, then the set val_i returned by an honest node $\mathcal{P}_i$ is not empty, and $|val_i| \leq 2$.*

Proof. If at least $n-2f$ honest nodes have the same input value v, then, according to Lemma 5, every honest node $\mathcal{P}_i$ can output a set val_i. Since $val_i \subseteq values_i$, and val_i is the set of values ω carried by $(\text{AUX}, \mathsf{id}, \omega)$ messages from at least $n-f$ distinct nodes, and there are at most f malicious nodes, some AUX messages must come from honest nodes. If the set val_i is an empty set, it is impossible because some honest nodes multicast a $(\text{AUX}, \mathsf{id}, \omega)$ where $\omega \in values_i$, and $values_i \neq \emptyset$. Hence, the set val_i returned by an honest node $\mathcal{P}_i$ is not empty. Following Lemma 2, there exist at most two different VAL messages among all honest nodes. According to the code, the set $values_i$ at most contains two values; since $val_i \subseteq values_i$, hence, $|val_i| \leq 2$. □

Lemma 7. Validity. *If at least $n-2f$ honest nodes have the same input value v, and $\mathcal{P}_i$ and $\mathcal{P}_j$ are honest nodes with $|val_i| \leq |val_j|$, then $val_i \subseteq val_j$. Additionally, if $|val_i| = 2$, then $v \in val_i$.*

Proof. Suppose $val_i = \{v\}$ and $val_j = \{v'\}$, where $v \neq v'$. Both val_i and val_j are the sets of values ω carried by $(\text{AUX}, \mathsf{id}, \omega)$ messages from at least $n-f$ distinct nodes. Since every honest node sends only one AUX message, if $v \neq v'$, it means that one honest node sent two different AUX messages. This is a contradiction, so we have $val_i = val_j$.

Following Lemma 2, there exist at most two different VAL messages among all honest nodes, namely $(\text{VAL}, \mathsf{id}, v)$ and $(\text{VAL}, \mathsf{id}, v')$. According to Algorithm 2, if $|val_i| = 2$, the unique possibility is $val_i = \{v, v'\}$. Suppose for honest node $\mathcal{P}_j$, it has $v'' \in val_j$, then at least one honest node multicasts $(\text{AUX}, \mathsf{id}, v'')$ message. It also means that at least one honest node multicasts $(\text{VAL}, \mathsf{id}, v'')$ message. If $v'' \neq v$ and $v'' \neq v'$, then it contradicts Lemma 2. Hence, if $|val_i| = |val_j| = 2$, then $val_i = val_j$ and $v \in val_i$. □

Theorem 2. *The Algorithm 2 implements a Synchronized Multi-valued Broadcast protocol (cf. Definition 2), in the presence of a computationally bounded and adaptive adversary who may corrupt up to $f < n/3$ parties.*

Proof. Lemma 4, 5, 6 and 7 complete the proof. □

Complexity analysis. According to Lemma 1, in the worst case, each honest node multicasts one FILTER message, two distinct FILTERECHO messages, at most three distinct VAL messages, and one AUX message. Each party will consider at most the maximum allowable number of messages received from other nodes. Suppose the input value v has size ℓ, then each type of message also has a size of $\mathcal{O}(\ell)$, resulting in the protocol having a message complexity of $\mathcal{O}(n^2)$ and a communication complexity of $\mathcal{O}(n^2\ell)$.

6 Hash Based MVBA

In this section, we present our hash based MVBA with adaptive security, leveraging only hash functions. Our approach attains subcubic communication complexity of $\mathcal{O}(\kappa(n\ell + \lambda n^2 \log n))$, and optimal fault tolerance with $n \geq 3f + 1$, where ℓ represents the size of the input values, κ is a statistical security parameter, λ denotes the size of a hash value, and n refers to the number of nodes.

6.1 Overview of our MVBA protocol

As shown in Fig.2 in the introduction, our construction revolves around the properties of the SMID, particularly focusing on its validity. The fundamental ideas of our MVBA protocol are as follows: all honest nodes participate in the SMID-dispersal instance to disperse their own input values, and they wait until the SMID-dispersal has been successfully completed. During this time, the entire network ensures that there exists a set I with at least $n/3$ indexes, and if all honest nodes with index $i(\in I)$ as input, then at least $n - 2f$ honest nodes can reconstruct the same value that satisfies the Predicate function. Afterward, a leader election protocol Election is invoked to randomly elect κ nodes $\{\mathcal{P}_{s_1}, \ldots, \mathcal{P}_{s_\kappa}\}$ such that one of the nodes belongs to the set I (for simplicity, we assume $s_i < s_j$ if $i < j$). Subsequently, all honest nodes participate in the κ SMID-recast subprotocols. For an honest node $\mathcal{P}_i$, if the corresponding SMID-recast[id](s_z) subprotocol returns a value, then $\mathcal{P}_i$ inputs the value into the SMB$[z]$. If the SMB$[z]$ returns a valid set vset (i.e., $|\mathsf{vset}| \leq 2$), then the element of vset is input to the corresponding ARC protocol. All honest nodes await the output of 2κ ARC instances, and if any ARC protocol produces an output, then 1 is input into the corresponding ABA. For an honest node $\mathcal{P}_i$, if any of the ABA instances outputs 1, then $\mathcal{P}_i$ inputs 0 into the remaining ABA instances that do not have an input. The termination condition for the entire protocol is the termination of the 2κ ABA instances.

6.2 Details of the MVBA protocol

The flow of our MVBA is outlined in Fig. 2. We now provide a detailed description. Specifically, an MVBA instance with identifier id proceeds as follows:

1. *SMID-dispersal phase* (lines 1-2). All honest nodes distribute their input values v_i to the entire network by invoking SMID-dispersal with v_i as the input.
2. *Election phase* (lines 3-4). Upon receiving (`disperse-done`, id) from the SMID-dispersal protocol, indicating that at least $n - 2f$ honest nodes have completed their dispersal, $\mathcal{P}_i$ will invoke Election to select κ distinct nodes.
3. *SMID-recast phase* (lines 5-6). If Election returns κ values $\{\mathcal{P}_{s_1}, \ldots, \mathcal{P}_{s_\kappa}\}$, then all honest nodes invoke SMID-recast to attempt to recast the inputs of these nodes.
4. *SMB phase* (lines 7-12). If an honest node $\mathcal{P}_i$ receives (`recast-output`, s_z, v_{s_z}) from SMID-recast for any $z \in [\kappa]$, and the value v_{s_z} satisfies the predicate Predicate, then $\mathcal{P}_i$ computes the corresponding vector commitment vc_z and position proofs $\{\pi_{z,j}\}_{j\in[n]}$. Subsequently, $\mathcal{P}_i$ input vc_z to SMB$[z]$.
5. *ARC phase* (lines 13-17). Suppose SMB returns vset to an honest node $\mathcal{P}_i$. If the size of vset is at most 2, then $\mathcal{P}_i$ considers the elements of vset as the input for ARC.

Algorithm 3 The MVBA protocol with identifier id for $\mathcal{P}_i$

Initialize $\mathsf{value}_z = \mathsf{vc}_z = \mathsf{vc}_{z,a} \leftarrow \perp$, $\pi_{z,i} = c_{z,i} \leftarrow \perp$, for $z \in [\kappa]$, $a \in \{1,2\}$, and $i \in [n]$; $\mathsf{terminate} \leftarrow 0$ and $\mathsf{Fragments} \leftarrow \emptyset$.

1: **upon** receiving input v_i s.t. $\mathsf{Predicate}(v_i) = true$ **do**
2: input v_i to SMID-dispersal[id] ▷ dispersal
3: **upon** SMID-dispersal[id] returns (`disperse-done`, id) **do**
4: $\{\mathcal{P}_{s_1}, \cdots, \mathcal{P}_{s_\kappa}\} \leftarrow \mathsf{Election}$ ▷ leader election: threshold $n - f$
5: **for** each $z \in \{1, 2, 3, \ldots, \kappa\}$ **do**
6: input s_z to SMID-recast[id] ▷ recast
7: **upon** SMID-recast[id] returns (`recast-output`, s_z, v_{s_z}) **do**
8: **if** $\mathsf{Predicate}(v_{s_z}) = 1$ **then**
9: Let $\boldsymbol{c}_z := [c_{z,1}, c_{z,2}, \ldots, c_{z,n}] \leftarrow \mathsf{EC.Encode}(v_{s_z}, n, f+1)$
10: $\mathsf{vc}_z \leftarrow \mathsf{VCom}(\boldsymbol{c}_z)$ and $\mathsf{value}_z \leftarrow v_{s_z}$
11: **for** $j \in [n]$, $\pi_{z,j} \leftarrow \mathsf{Open}(\mathsf{vc}_z, c_j, j)$
12: input vc_z to SMB[z] ▷ SMB
13: **upon** SMB[z] returns vset **do**
14: **if** $\mathsf{vset} := \{\mathsf{vc}_{z,1}\}$ **then**
15: input $\mathsf{vc}_{z,1}$ to ARC[$z, 1$] and input $\mathsf{vc}_{z,1}$ to ARC[$z, 2$] ▷ ARC
16: **if** $\mathsf{vset} := \{\mathsf{vc}_{z,1}, \mathsf{vc}_{z,2}\}$ **then** ▷ sorting with alphabetical order
17: input $\mathsf{vc}_{z,1}$ to ARC[$z, 1$] and input $\mathsf{vc}_{z,2}$ to ARC[$z, 2$]
18: **upon** ARC[z, a] returns $\mathsf{vc} \wedge z \in [\kappa] \wedge a \in [2]$ **do**
19: $\mathsf{vc}_{z,a} \leftarrow \mathsf{vc}$
20: **if** no input has been provided to ABA[z, a] **then**
21: input 1 to ABA[z, a] ▷ ABA
22: **upon** any ABA[z', a'] returns $1 \wedge z' \in [\kappa] \wedge a' \in [2]$ **do**
23: **for** any $z \in \{1, 2, 3, \ldots, \kappa\}$ and $a \in \{1, 2\}$ **do**
24: **if** no input has been provided to ABA[z, a] **then**
25: input 0 to ABA[z, a]
26: **upon** all 2κ ABA instances have completed **do**
27: Let (z^*, a^*) be the smallest pair such that ABA[z^*, a^*] outputs 1
28: **if** $\mathsf{vc}_{z^*,a^*} = \perp$ **then**
29: **wait** until ARC[z^*, a^*] returns vc
30: $\mathsf{vc}_{z^*,a^*} \leftarrow \mathsf{vc}$
31: **if** $\mathsf{vc}_{z^*} = \mathsf{vc}_{z^*,a^*}$ **then**
32: **send** (`fragment`, id, $\mathsf{vc}_{z^*}, c_{z^*,j}, \pi_{z^*,j}$) to $\mathcal{P}_j$ for every $j \in [n]$
33: **upon** receiving (`fragment`, `sid`, $\mathsf{vc}', c_i', \pi_i'$) such that $\mathsf{vc}_{z^*,a^*} = \mathsf{vc}'$ **do**
34: **if** $\mathsf{VerifyOpen}(\mathsf{vc}', c_i', i, \pi_i') = 1$ and have not send `forward` message **then**
35: **send** (`forward`, `sid`, $\mathsf{vc}', c_i', \pi_i'$) to all parties and $\mathsf{terminate} = 1$
36: **upon** receiving (`forward`, `sid`, $\mathsf{vc}', c_j', \pi_j'$) from $\mathcal{P}_j$ for the first time **do**
37: **if** $\mathsf{vc}' = \mathsf{vc}_{z^*,a^*}$ and $\mathsf{VerifyOpen}(\mathsf{vc}', c_j', j, \pi_j') = 1$ **then**
38: $\mathsf{Fragments} \leftarrow \mathsf{Fragments} \cup \{(j, c_j')\}$
39: **upon** $|\mathsf{Fragments}| = f + 1$ **do**
40: $\mathsf{value}_{z^*} \leftarrow \mathsf{EC.Decode}(\mathsf{Fragments}, n, f+1)$ ▷ recovery
41: **wait** until $\mathsf{value}_{z^*} \neq \perp$ and $\mathsf{terminate} = 1$
42: **output** value_{z^*} ▷ output

6. *ABA phase* (lines 18-25). If ARC[z, a] returns vc to $\mathcal{P}_i$, then $\mathcal{P}_i$ invokes ABA[z, a] with 1 as input, provided that he has not already done so. Furthermore, if $\mathcal{P}_i$ receives an output of 1 from any ABA instance, then for any $z' \in [\kappa]$ and $a' \in \{1, 2\}$, $\mathcal{P}_i$ invokes ABA[z', a'] with input 0 if he has not yet provided an input to these ABA instance.

7. *Output phase* (lines 26-42). Wait until all 2κ ABA instances output their values, meaning that all 2κ ABA instances have terminated. Suppose (z^*, a^*) is the smallest pair such that ABA[z^*, a^*] outputs 1.

Then, an honest node $\mathcal{P}_i$ waits for the output vc_{z^*,a^*} from $\mathsf{ARC}[z^*, a^*]$. It's possible that some honest nodes have not received the corresponding value of vc_{z^*,a^*}. To help these nodes receive it, two additional types of messages are introduced: `fragment` and `forward`. Specifically, if some honest nodes have the corresponding value, they send a `fragment` to every node. If a node receives a valid `fragment` message for the first time, it sends a `forward` message to all nodes. Then, if some honest nodes have not received the value, they will wait for $f + 1$ valid `forward` messages from distinct nodes. After decoding, they output the decoded value.

6.3 Security and Complexity analysis

We first establish the following lemma to show that those κ indices returned by Election (at Line 4, Algorithm 3) contain a "good" index with probability of at least $1 - 2^{-\Omega(\kappa)}$.

Lemma 8. *Suppose there is a set $\mathbb{P} = \{1, \ldots, n\}$. For any subset $\mathbb{I} \subseteq \mathbb{P}$ such that $|\mathbb{I}| \geq f + 1$, the* Election *selects κ nodes from $\mathbb{P}$. Then, except with probability $\exp(-\Omega(\kappa))$, at least one selected node has an index in $\mathbb{I}$.*

Proof. When the Election selects κ nodes uniformly at random from $\mathbb{P}$, the probability that none of these nodes belong to $\mathbb{I}$ is the number of ways to choose all κ nodes from the $2f$ nodes not in $\mathbb{I}$ divided by the total number of ways to choose κ nodes from $n = 3f + 1$ nodes. That is, $\Pr[\text{none in } \mathbb{I}] = \frac{\binom{2f}{\kappa}}{\binom{3f+1}{\kappa}}$.

Therefore, the probability that at least one selected node has an index in $\mathbb{I}$ is given by: $\Pr[\text{at least one in } \mathbb{I}] = 1 - \frac{\binom{2f}{\kappa}}{\binom{3f+1}{\kappa}} \geq 1 - \left(\frac{2}{3}\right)^{\kappa} = 1 - exp(-\Omega(\kappa))$. □

Choices of κ. We now determine the parameter κ required to achieve 40-bit and 80-bit statistical security (which correspond to 128-bit and 256-bit computational security, respectively). Specifically, we seek the smallest integer κ such that $\Pr[\text{at least one in } \mathbb{I}] \geq 1 - 2^{-\gamma}$ for $\gamma \in \{40, 80\}$. Solving this inequality yields $\kappa = 69$ for 40-bit security and $\kappa = 137$ for 80-bit security.

Lemma 9. Termination. *If every honest node p_i inputs an externally valid value v_i, then every honest node outputs a value.*

Proof. In MVBA, every honest node inputs a value, and according to the *termination* property of SMID, every honest node can complete the SMID-dispersal. Consequently, all honest nodes will invoke Election during the election phase. Following the functionality of Election, all honest nodes will receive the same κ indices of selected nodes.

According to the *validity* property of SMID, there exists a subset $\mathbb{I}$ of initially honest nodes such that $|\mathbb{I}| \geq f + 1$. For $i \in \mathbb{I}$, then at least $n - 2f$ honest nodes can reconstruct the initial input value v_i of $\mathcal{P}_i$. Following Lemma 8, among the κ selected nodes, except with probability $\exp(-\Omega(\kappa))$, at least one selected node (say $\mathcal{P}_i$) will have an index belonging to $\mathbb{I}$. This also implies that at least $f + 1$ honest nodes will have the same input value v_i for the same SMB instance with identifier i in the SMB phase. In this case, following the *obligation* property of SMB, there will exist two values, v'_1 and v'_2, such that every honest node $\mathcal{P}_i$ will output a set $val_i \subseteq \{v'_1, v'_2\}$ from the SMB instance with identifier i.

Because every honest node $\mathcal{P}_j$ holds a set $val_j \subseteq \{v'_1, v'_2\}$ corresponding to identifier i, the following cases must occur: (1) all honest nodes input v'_1 into the $\mathsf{ARC}[i, 1]$; or (2) all honest nodes input v'_2 into the $\mathsf{ARC}[i, 2]$. Following the *validity* property of ARC, all honest nodes can receive the same output from at least one ARC instance, which also implies that all honest nodes will input 1 into the corresponding ABA instance. For honest nodes $\mathcal{P}_i$, once an ABA instance returns 1, they will input 0 into all other ABA instances if they have not yet provided inputs. Then, according to the *termination* of ABA, all ABA instances can be completed.

The *agreement* property of ABA ensures that all honest nodes obtain the same output from these ABA instances; therefore, they will all attempt to recover the corresponding value. Considering the *agreement* and *justification* properties of ARC, all honest nodes obtain the same vc, which originates from some honest

node's input. By the protocol design, this also implies that vc is the output of SMB. Following the *justification* property of SMB, at least one honest node must have provided vc as input to SMB. According to Algorithm 3, at least one honest node possesses a value v such that v corresponds to the vector commitment vc. Hence, this honest node will send `fragment` messages to all nodes, prompting every honest node to subsequently send a `forward` message to all nodes. As a result, each honest node will receive at least $f+1$ valid `forward` messages, allowing them to successfully decode and output a value. □

Lemma 10. ***Agreement.*** *Any two distinct honest nodes always output the same value.*

Proof. From the above analysis, all honest nodes will obtain the same vc and attempt to recover the corresponding value. Moreover, at least one honest node (say $\mathcal{P}_i$) possesses the corresponding value v. According to Algorithm 3 lines 32-40, $\mathcal{P}_i$ sends a `fragment` message to each $\mathcal{P}_j \in [n]$ together with the corresponding fragment. Once $\mathcal{P}_j$ receives a valid `fragment` message, it forwards the message to all nodes via `forward` messages. As a result, all honest nodes can wait for $f+1$ valid `forward` messages associated with the vector commitment vc, and then proceed to decode the fragments. By the *correctness* properties of both the erasure code and the vector commitment, all honest nodes will successfully decode the same value. □

Lemma 11. ***External-validity.*** *For any honest output v,* Predicate$(v) = 1$.

Proof. From the above analysis, the vector commitment of the output value corresponds to some honest node's input in SMB. According to Algorithm 3, an honest node will input the corresponding vector commitment to SMB only if the output value v satisfies Predicate$(v) = 1$. □

Theorem 3. *Assuming the underlying hash function is collision resistant, the Algorithm 3 implements* MVBA*, in the presence of a computationally bounded and adaptive adversary who may corrupt up to $f < n/3$ nodes.*

Proof. Lemma 9, 10 and 11 complete the proof. □

Complexities analysis. In Algorithm 3, illustrated in Figure 2, all honest nodes engage in one SMID-dispersal, one election protocol Election, and subsequently, κ instances of SMID-recast, κ instances of SMB, 2κ instances of ARC, and 2κ instances of ABA. The input size of SMID-dispersal is ℓ, while the input size of SMB and ARC is λ. Throughout this paper, we assume the use of an ideal common coin to implement Election, and thus, we omit its associated cost. The cost performance of a single instance/phase is as follows:

1. Both SMID-dispersal and SMID-recast have a message complexity of $\mathcal{O}(n^2)$ and a communication complexity of $\mathcal{O}(n\ell + \lambda n^2 \log n)$.
2. Both SMB and ARC exhibit the same cost performance, with a message complexity of $\mathcal{O}(n^2)$ and communication complexity of $\mathcal{O}(\lambda n^2)$.
3. Each ABA instance has a message complexity of $\mathcal{O}(n^2)$ and communication complexity of $\mathcal{O}(n^2)$.
4. In the output phase, each honest node sends n `fragment` messages and n `forward` messages, where the size of both `fragment` and `forward` messages is $\mathcal{O}(\lambda + \ell/n)$. Consequently, the total number of exchanged messages is $\mathcal{O}(n^2)$, and the cost of bit communication is $\mathcal{O}(n\ell + \lambda n^2 \log n)$.

As a result, the overall message complexity of MVBA is $\mathcal{O}(\kappa n^2)$, and the communication complexity is $\mathcal{O}(\kappa(n\ell + \lambda n^2 \log n))$, which is dominated by the κ instances of SMID-recast.

6.4 MVBA with Fewer ABA Instances

For ease of presentation, our hash-based MVBA protocol in Section 6.2 employs $2k$ ABA instances, which requires at least $\mathcal{O}(\kappa)$ common coin invocations. In this section, we introduce a novel approach that reduces the number of common coin invocations from $\mathcal{O}(\kappa)$ to $\mathcal{O}(\log \kappa)$.

Recall that in the MVBA protocol of Section 6, the 2κ ABA instances are used to "check" the status of all 2κ ARC instances, so that all nodes can agree on the same "valid" ARC and take its output. Intuitively, this process resembles a linear search algorithm.

We now introduce a "binary-search-like" strategy to select a valid ARC instance. Specifically, we first divide the 2κ ARC instances into two halves. The *first half* consists of

$$\{\mathsf{ARC}[1,1], \mathsf{ARC}[1,2], \ldots, \mathsf{ARC}[\kappa/2,1], \mathsf{ARC}[\kappa/2,2]\},$$

and the *second half* consists of

$$\{\mathsf{ARC}[\kappa/2+1,1], \mathsf{ARC}[\kappa/2+1,2], \ldots, \mathsf{ARC}[\kappa,1], \mathsf{ARC}[\kappa,2]\}.$$

We then proceed iteratively:

- First, we run two ABA instances. If any ARC instance in the first half outputs, we input 1 to the first ABA (if it has not already received input). Similarly, if one ABA outputs while the other has not yet received input, then we input 0 to the latter.
- If the first ABA outputs 1, we recursively divide the first half of ARC instances into two groups.
- If instead the first ABA outputs 0 and the second ABA outputs 1, we recursively divide the second half of ARC instances.

By repeating this process, all nodes eventually agree on the same output from the same ARC instance with $\mathcal{O}(\log \kappa)$ ABA instances, instead of $\mathcal{O}(\kappa)$. For completeness, we also describe the optimization of MVBA in Algorithm 4.

Security intuition. The Algorithm 4 essentially follows the same idea as the MVBA protocol in Section 6.2. Following the same argument as in Lemma 9, all honest nodes receive the same output from at least one ARC instance, which in turn implies that they will all input 1 into the corresponding ABA instance. Moreover, if an honest node inputs 0 to an ABA instance, then according to Algorithm 4, this implies that the node has already received a 1 from another ABA instance. Furthermore, if some ABA outputs 1, it indicates that some honest node has input 1 into that ABA, which in turn implies that honest nodes have received output from some ARC in the corresponding interval. By the totality and agreement properties of ARC, all honest nodes can receive the same value from the same ARC instance. Therefore, by modifying the execution in ABA, we can guarantee the same functionality as the original construction described in Section 6.2.

Comparing our two MVBA variants. We now have two variants of MVBA, algorithm 3 and algorithm 4, which we compare here. They share the same communication complexity in the coin-aided model, where we do not account for the cost of the common coin. Their main difference lies in the number of common coins required. In the general case, the MVBA protocol in Section 6.2 requires $\mathcal{O}(\kappa)$ ABA instances in parallel, and hence $\mathcal{O}(\kappa)$ common coins, incurring $O(\log \kappa)$ rounds. The optimized MVBA requires only $\mathcal{O}(\log \kappa)$ ABA instances (and thus $\mathcal{O}(\log \kappa)$ common coins), but due to the nature of binary search, it still incurs $O(\log \kappa)$ rounds. On the other hand, in the fault-free case under a synchronous network, if we employ Crain's ABA protocol [31], which enjoys the *good-case-coin-free* property, the complexity of the MVBA protocol in Section 6.2 becomes even more efficient. This property guarantees that if all honest nodes input the same value into an ABA, they can reach agreement without invoking the common coin subprotocol. In this optimistic case, the time complexity of the entire protocol reduces to $\mathcal{O}(1)$, whereas the protocol in algorithm 4 still requires $O(\log \kappa)$ rounds.

6.5 MVBA with Information-Theoretic Security

The MVBA protocol in Algorithm 3 assumes a collision-resistant hash function, as required by the underlying vector commitment scheme (used both in SMID and directly in Algorithm 3), while the other components are already information-theoretically (IT) secure. In this section, we present an IT-secure SMID construction that removes the need for a cryptographic hash, at the cost of an increased communication complexity of $\mathcal{O}(n^2\ell)$. Based on this construction, we slightly modify Algorithm 3, yielding an IT-secure MVBA protocol with communication complexity $\mathcal{O}(\kappa n^2 \ell)$.

IT-secure SMID. We describe the protocol in Algorithm 5. In this protocol, each node disperses its input message by directly *multicasting* the entire message to the network. Then, after two rounds of Ok

Algorithm 4 The optimized MVBA protocol with an identifier id for $\mathcal{P}_i$

Initialize $\mathsf{value}_z = \mathsf{vc}_z = \mathsf{vc}_{z,a} \leftarrow \perp$, $\pi_{z,i} = c_{z,i} \leftarrow \perp$, for $z \in [\kappa]$, $a \in \{1,2\}$, and $i \in [n]$; $\mathsf{terminate} \leftarrow 0$ and $\mathsf{Fragments} \leftarrow \emptyset$.

1: same as lines 1–19 of Algorithm 3
2: **execute** $\mathsf{ProcessLevel}(1, 1, \kappa)$
3: **upon** $\mathsf{ProcessLevel}(1, 0, \kappa)$ returns vc_{z^*,a^*} **do**
4: **if** $\mathsf{vc}_{z^*} = \mathsf{vc}_{z^*,a^*}$ **then**
5: **send** $(\texttt{fragment}, \mathsf{id}, \mathsf{vc}_{z^*}, c_{z^*,j}, \pi_{z^*,j})$ to $\mathcal{P}_j$ for every $j \in [n]$
6: **upon** receiving $(\texttt{fragment}, \texttt{sid}, \mathsf{vc}', c'_i, \pi'_i)$ such that $\mathsf{vc}_{z^*,a^*} = \mathsf{vc}'$ **do**
7: **if** $\mathsf{VerifyOpen}(\mathsf{vc}', c'_i, i, \pi'_i) = 1$ and have not send `forward` message **then**
8: **send** $(\texttt{forward}, \texttt{sid}, \mathsf{vc}', c'_i, \pi'_i)$ to all parties and $\mathsf{terminate} = 1$
9: **upon** receiving $(\texttt{forward}, \texttt{sid}, \mathsf{vc}', c'_j, \pi'_j)$ from $\mathcal{P}_j$ for the first time **do**
10: **if** $\mathsf{vc}' = \mathsf{vc}_{z^*,a^*}$ and $\mathsf{VerifyOpen}(\mathsf{vc}', c'_j, j, \pi'_j) = 1$ **then**
11: $\mathsf{Fragments} \leftarrow \mathsf{Fragments} \cup \{(j, c'_j)\}$
12: **upon** $|\mathsf{Fragments}| = f + 1$ **do**
13: $\mathsf{value}_{z^*} \leftarrow \mathsf{EC.Decode}(\mathsf{Fragments}, n, f+1)$ ▷ recovery
14: **wait** until $\mathsf{value}_{z^*} \neq \perp$ and $\mathsf{terminate} = 1$
15: **output** value_{z^*} ▷ output

function $\mathsf{ProcessLevel}(s, L, R)$:
16: **if** $s < \lfloor \log \kappa \rfloor$ **then** ▷ $s < \lfloor \log \kappa \rfloor$
17: **wait** until $\exists\ \mathsf{vc}_{z,a} \neq \perp$ and $z \in [L, \lfloor (L+R)/2 \rfloor]$
18: **if** hasn't provided input to $\mathsf{ABA}[s,1]$ **then** input 1 to $\mathsf{ABA}[s,1]$
19: **wait** until $\exists\ \mathsf{vc}_{z,a} \neq \perp$ and $z \in [\lceil (L+R)/2 \rceil, R]$
20: **if** hasn't provided input to $\mathsf{ABA}[s,2]$ **then** input 1 to $\mathsf{ABA}[s,2]$
21: **upon** any $\mathsf{ABA}[s,a']$ returns 1 and $a' \in \{1,2\}$ **do**
22: **if** hasn't provided input to $\mathsf{ABA}[s,a]$ for $a \in \{1,2\}$ **then**
23: input 0 to $\mathsf{ABA}[s,a]$
24: **if** $\mathsf{ABA}[s,1]$ outputs 1 **then**
25: $\mathsf{ProcessLevel}(s+1, L, \lfloor (L+R)/2 \rfloor)$
26: **if** $\mathsf{ABA}[s,1]$ outputs 0 and $\mathsf{ABA}[s,2]$ outputs 1 **then**
27: $\mathsf{ProcessLevel}(s+1, \lceil (L+R)/2 \rceil, R)$
28: **if** $s \geq \lfloor \log \kappa \rfloor$ **then** ▷ $s \geq \lfloor \log \kappa \rfloor$
29: **upon** $\exists\ \mathsf{vc}_{z,a} \neq \perp$ and $L \leq z \leq R$ **do**
30: **if** hasn't provided input to $\mathsf{ABA}[s,z,a]$ **then**
31: input 1 to $\mathsf{ABA}[s,z,a]$
32: **upon** any $\mathsf{ABA}[s,z',a']$ returns 1 and $L \leq z' \leq R$ and $a' \in [2]$ **do**
33: **if** hasn't provided input to $\mathsf{ABA}[s,z,a]$ for $a \in \{1,2\}$ and $L \leq z \leq R$ **then**
34: input 0 to $\mathsf{ABA}[s,z,a]$
35: **upon** all $2(R-L+1)$ ABA instances have completed **do**
36: Let $(z \in [L,R], a \in \{1,2\})$ be the smallest pair s.t. $\mathsf{ABA}[s,z,a]$ outputs 1
37: **if** $\mathsf{vc}_{z,a} = \perp$ **then**
38: **wait** until $\mathsf{ARC}[z,a]$ returns vc
39: $\mathsf{vc}_{z^*,a^*} \leftarrow \mathsf{vc}$
40: return vc_{z^*,a^*}

and COMPLETED messages (as in Algorithm 1), all nodes can terminate the dispersal phase. In the recast phase, each node outputs the input value if it has received this value before terminating the dispersal. No communication occurs during the recast phase.

It is straightforward to verify that this SMID protocol satisfies all security properties stated in Definition 1. We summarize this result in the following lemma.

Algorithm 5 The IT-SMID protocol with identify id for $\mathcal{P}_i$

Initialize: $values[j] \leftarrow \perp$ for $j \in [n]$

▷ SMID-dispersal[id](v_i)

1: **upon** receiving input v_i **do**
2: **multicast** (VALUE, v_i)
3: **upon** receiving (VALUE, v_j) from $\mathcal{P}_j$ for the first time **do**
4: $values[j] \leftarrow v_j$
5: **send** OK to $\mathcal{P}_j$
6: **upon** receiving OK from $\mathcal{P}_j$ for the first time **do**
7: $S_{ok} \leftarrow S_{ok} \cup \{j\}$
8: **if** $|S_{ok}| = n - f$ **then**
9: **multicast** COMPLETED to all
10: **upon** receiving COMPLETED from $\mathcal{P}_j$ for the first time **do**
11: $S_{comp} \leftarrow S_{comp} \cup \{j\}$
12: **if** $|S_{comp}| = n - f$ **then**
13: **return** (`dispersal-done`, id)

14: **upon** receiving input s **do** ▷ SMID-recast[id](s)
15: **if** $values[s] = v_s$ **then**
16: **return** (`recast-output`, s, v_s)

Lemma 12. *Algorithm 5 implements an* SMID *protocol (cf. Definition 1) in the presence of a computationally unbounded adaptive adversary that may corrupt up to $f < n/3$ nodes.*

IT-secure MVBA. With the IT-secure SMID as a component, we can obtain an IT-secure MVBA by applying the following modifications to Algorithm 3:

- In Lines 9–12, directly input the entire message v_{s_z} to SMB$[z]$. Equivalently, set $vc_{s_z} \leftarrow v_{s_z}$ and provide vc_{s_z} to SMB$[z]$.
- After Line 30, where $vc_{z^*,a^*} \leftarrow vc$, directly return vc_{z^*,a^*}. All subsequent operations are omitted.

The optimization technique in Section 6.4 can be applied to the IT-secure SMID without any modification. The security proof follows directly from the proof of Theorem 3. Regarding efficiency, the SMID phase incurs $O(n^2\ell)$-bit communication. The 2κ instances of SMB with ℓ-bit inputs incur $O(\kappa n^2\ell)$-bit communication, and the 2κ instances of ARC also incur $O(\kappa n^2\ell)$-bit communication. Finally, the $\log \kappa$ instances of ABA incur $O(n^2 \log \kappa)$-bit communication. Therefore, the overall communication complexity is $O(\kappa n^2\ell)$. We summarize the result in the following theorem.

Theorem 4. *Following the above modifications, the adjusted protocol implements* MVBA *with expected communication complexity $O(\kappa n^2\ell)$. This holds in the presence of an unbounded adaptive Byzantine adversary that may corrupt up to $f < n/3$ nodes.*

7 Hash-based ACS and Asynchronous MPC

In this section, we present a direct application of our MVBA to ACS, and then show how we can improve the state-of-art asynchronous MPC using our ACS.

7.1 Optimal Asynchronous Common Subset from MVBA

Our ACS essentially follows Cachin et al.'s framework [17], which builds an ACS protocol from a MVBA protocol using digital signatures (and thus assuming a bare PKI). In the framework, each node firstly

multicasts its input value along with its signature, then after collecting $n-f$ signed values from distinct nodes, invokes an MVBA with the collected signed values to agree on one vector. It is easy to argue that the elegant construction satisfies all security properties of ACS. At a high level, the *termination* and *agreement* of MVBA guarantee the *totality* and *agreement* of ACS, while the *external validity* of MVBA plus the *unforgeability* of the underlying signature ensures the *validity* of ACS.

Theorem 5. *Let* ACS *be a protocol instantiated from [17] using our hash-based* MVBA *in Algorithm 3 and any hash-based signature scheme with existential unforgeability. Assuming the underlying hash function is collision-resistant, under a PKI setup, and in the coin-aided model, the* ACS *protocol satisfies agreement, validity, and totality. It has a communication complexity of* $O(\kappa n^2\ell + \kappa\lambda n^2 \log n)$ *and a round complexity of* $O(\log \kappa)$*, where* ℓ *is the bit-length of an input.*

The security analysis directly follows from [17]. Regarding communication complexity, the multicast phase incurs $O((\ell+\lambda)n^2)$ bits of communication for n nodes, each with an ℓ-bit input. The input to MVBA consists of $n-f$ values along with their corresponding signatures, resulting in a total size of $O((\ell+\lambda)n)$. Consequently, the overall communication complexity is $O(\kappa(\ell n^2 + \lambda n^2 \log n))$.

In terms of round complexity, the ACS protocol introduces only one additional round of multicast beyond the MVBA protocol. Therefore, the round complexity remains dominated by the MVBA protocol, which is $O(\log \kappa)$.

7.2 Applications to Asynchronous MPC

Next, we show how our ACS could solve the input agreement problem in AMPC.

Overview of offline-online AMPC. Almost all existing AMPC protocols [6, 9, 13, 20, 23, 25–27, 51] follow the offline-online paradigm. As the name suggests, the offline phase is independent of the function and the data to be evaluated and thus can be executed before the actual computation. The correlated randomness generated in the offline phase will be utilized in the subsequent online phase. The online phase will be invoked when the data and the computation task are known, which consists of an **input sub-phase** and a **function evaluation sub-phase**. First, each node $\mathcal{P}_i$ inputs a private value x_i and distributes secret shares of x_i to others. After that, the honest nodes should have consistent secret shares of the same set of $n-f$ input values. Then, in the function evaluation phase, the nodes jointly evaluate a function f on the shared inputs $(x_1, \ldots, x_n)$ and finally obtain a set of consistent secret shares on $f(x_1, \ldots, x_n)$.

In general, as the offline phase can be executed prior to the actual computation, it can accept an expensive offline protocol. On the other hand, the online phase, including both the input phase and the function evaluation phase, is always expected to be efficient. Moreover, there is a lot of work pushing down the communication complexity of the functional evaluation phase. Notably, in the information-theoretical setting, we know solutions to evaluate a function f (modeled as a circuit C) at the cost of $\mathcal{O}\big(M \cdot n + D \cdot n^2\big)$ [25–28], where M is the number of multiplicative gates in the circuit C, n is the number of nodes, and D is the depth of the circuit. Existing AMPC protocols apply n instances of RBC [16] or asynchronous completed secret sharing [28] to disseminate the private inputs, causing $\Omega(n^3)$ communication cost already; An agreement on core set protocol is further employed to decide which $n-f$ broadcast instances have terminated, incurring another $\mathcal{O}\big(n^3\big)$ communication cost. In summary, the current input agreement employed in existing AMPC constitutes a bottleneck in terms of communication complexity, particularly when the number of multiplicative gates to be evaluated is not very large, e.g., $\mathcal{O}\big(n^2\big)$.

The INPUT protocol and its realization from ACS. We focus on the Shamir sharing, as it is employed in almost all AMPC protocols with a liveness guarantee. The goal of the input phase w.r.t. the Shamir secret sharing is to enable each honest P_i to distribute its private input v_i to the network, such that every honest P_i terminates with a set of secret shares $\{v_j^{(i)}\}_{j\in S}$, where $S \subset [n]$, $|S| = n-f$. In particular, the shares received by different honest nodes for the same j are consistent; if P_j is forever honest, $\{v_j^{(i)}\}_{i\in \mathsf{Honest}}$ are Shamir secret shares of the input of P_j. Regarding secrecy, the adversary should only know the input length. As the input phase is a naturally a part of AMPC, we formalize the above goal as an ideal functionality $\mathcal{F}_{\mathsf{Input}}$, which is provided in Fig.12 in Appendix G.

Algorithm 6 The INPUT protocol with identifier id for $\mathcal{P}_i$

Preprocessing: For each $i \in [n]$, uniformaly sample r_i from the input space, and generate $(r_{i,1}, \ldots, r_{i,n}) \leftarrow \mathsf{SS}(r_i)$. Send $(r_i, (r_{j,i})_{j\in[n]})$ to each $\mathcal{P}_i$.

1: **upon** receiving input x_i **do** ▷ For any $\mathcal{P}_i$
2: Compute $\bar{x}_i = r_i + x_i$
3: input $(\bar{x}_i)$ to ACS[id]
4: **upon** ACS[id] returns comonSet **do**
5: Parse commonSet $= \{(j, \bar{x}_j)\}_{j\in\mathbb{J}}$
6: Compute $x_{j,i} = \bar{x}_j - r_{j,i}$, for each $j \in \mathbb{J}$
7: **output** $(\mathbb{J}, \{x_{j,i}\}_{j\in\mathbb{J}})$.

In our construction, we leverage Choudhury et al.'s [24] trick of using a preshared random value r to mask the real input x, such that each node can obtain a secret share of x by combining the share of r and the masked $\bar{x}$. The masked value $\bar{x}$ will not leak any information about x, and thus we can employ ACS and let all honest nodes agree on a set of masked values. Remark that the input protocol can be easily generalized to AMPC protocols not based on Shamir's secret sharing. For example, in the threshold fully homomorphic encryption based MPC, we can use ACS to let all nodes agree on a set of ciphertexts such that further computation can be carried out on them. We present an input phase protocol in Algorithm 6. In Appendix G, we formally prove our INPUT protocol UC-realizes the ideal functionality $\mathcal{F}_{\mathsf{Input}}$, which can thus be plugged in the existing Shamir Secret Sharing based AMPC protocols to improve the efficiency of their input phase.

Instantiation and complexity analysis. Using our ACS in Theorem 5 to instantiate INPUT gives us an input protocol under the PKI setup. Particularly, since we already assume the correlated randomness, the coin assumed by the ACS essentially comes "for free". I.e., the nodes can simply reconstruct a pre-shared random value as the coin.

Combining the existing function evaluation protocols with our input protocol, we have an online-phase AMPC protocol under the PKI setup against adaptive Byzantine adversaries which corrupts up to $n/3$ nodes, with the communication complexity of $\mathcal{O}\big(Mn + Dn^2 + \kappa\lambda n^2 \log n\big)$, where M is the number of multiplicative gates in the circuit C, n is the number of nodes, and D is the depth of the circuit.

8 Efficiency Analysis

In this section, we provide a concrete efficiency analysis of our proposed MVBA and ACS protocols. We evaluate their performance under 128-bit and 256-bit security levels and compare them against the state-of-the-art hash-based designs, FIN-MVBA and FIN-ACS [36].

8.1 Analytical Cost Model

To quantify communication costs, we categorize message content into five primary types: (1) **Control information (C)**, including session IDs and message headers; (2) **Hash output (H)**, generated by the underlying hash function; (3) **Signature (S)**, produced by the digital signature scheme; (4) **Input payload (L)**, representing the input value of an honest node; and (5) **Erasure code fragment**, a segment of the input encoded via an erasure coding scheme. For messages carrying only a single binary value, we approximate the size as C. We utilize an $(n, f+1)$ erasure coding scheme, where the optimal fragment size is $L/(f+1)$. For simplicity in our analysis, we approximate the ratio f/n as $1/3$ and $(n-f)/n$ as $2/3$. We analyze communication costs by calculating the expected total data volume transmitted by honest parties. For components like SMB, where the number of messages depends on the adversarial strategy and input cases, we consider the worst-case scenario (i.e., the maximal number of messages an honest party might send). For the underlying ABA, we adopt the state-of-the-art design from [65]. When the protocol invokes a common coin, we estimate the cost as n^2C (representing each node multicasting a small message), consistent with the standard practice of reconstructing a pre-generated random value during the online phase.

Next, to facilitate the comparison, we first calculate the costs of the individual components and then sum them to derive the total cost of our MVBA (i.e., the optimized variant in Section 6.4) and ACS protocols.

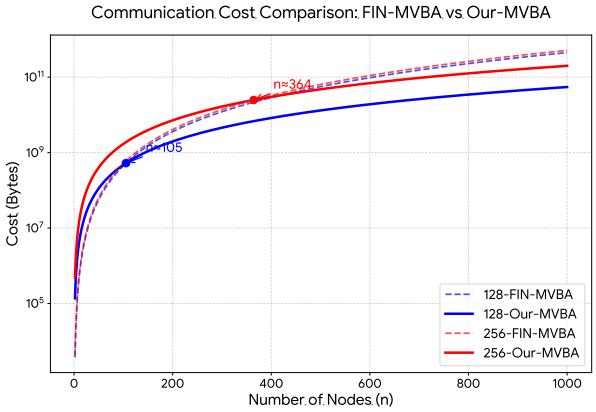

Fig. 3. Communication cost comparison between FIN-MVBA and Our MVBA

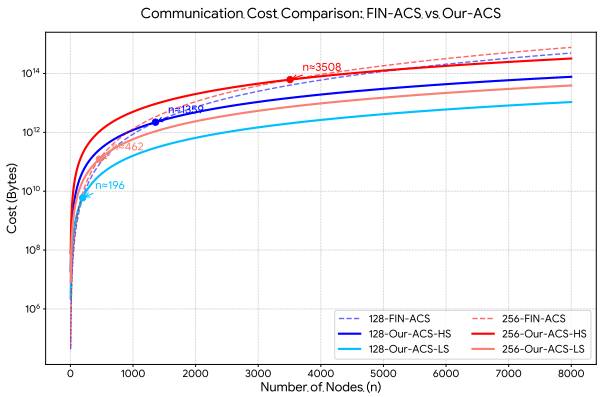

Fig. 4. Communication cost comparison between FIN-ACS and Our ACS.

Subsequently, we apply the same methodology to FIN-MVBA and FIN-ACS[14]. The total protocol costs are listed below, with a detailed component analysis provided in the Appendix A.

- $\mathsf{Cost}_{\mathsf{Our\text{-}MVBA}} = nL(9+3\kappa)+n^2(C(6+12\kappa+16\log\kappa)+H(3\log n+\kappa\log n+11\kappa))$.
- $\mathsf{Cost}_{\mathsf{Our\text{-}ACS}} = n^2(L+S+C)(7+2\kappa)+n^2(C(6+12\kappa+16\log\kappa)+H(3\log n+\kappa\log n+11\kappa)$.
- $\mathsf{Cost}_{\mathsf{FIN\text{-}MVBA}} = 2n^2L+5n^2C+n^3(2H+50C)$.
- $\mathsf{Cost}_{\mathsf{FIN\text{-}ACS}} = n^2(6C+H\log n+3L)+n^3(53C+4H+H\log n+\frac{4}{3}\log n)+3nL$.

8.2 Concrete Comparison

In this section, we compare the communication costs of our MVBA, our ACS, FIN-MVBA, and FIN-ACS protocols.

Configuration and Parameters. We evaluate the protocols using the following parameterized configuration format:

$$\mathsf{Security\ Level} - \mathsf{Protocol\ Name} - \mathsf{Signature\ Type}$$

Here, Protocol Name refers to the specific protocol (FIN-MVBA, FIN-ACS, Our-MVBA, or Our-ACS). The Signature Type parameter applies specifically to our ACS, where HS stands for the hash-based signature scheme SPHINCS [10], and LS stands for the lattice-based signature scheme Falcon [41]. Although LS is not hash-based, it offers post-quantum security, a priority for many applications, along with significantly smaller signature sizes than HS. For Security Level $= 128$, we set $H = 32$ Bytes (SHA-256) and $\kappa = 69$ (achieving 40-bit statistical security, comparable to 128-bit computational security). regarding the signature size S: if Signature Type $=$ HS, we set $S = 7,856$ Bytes [10]; if Signature Type $=$ LS, $S = 690$ Bytes. For Security Level $= 256$, we set $H = 64$ Bytes (SHA-512) and $\kappa = 137$ (achieving 80-bit statistical security, comparable to 256-bit computational security). Regarding the signature size S: if Signature Type $=$ HS, we set $S = 17,088$ Bytes [10]; if Signature Type $=$ LS, $S = 1330$ Bytes.

Finally, we justify our choices for control information and input size. We set the control information size $C = 8$ Bytes, which is sufficient to encode a unique session identifier and message type tags with negligible collision probability. For the input size L, we adopt two distinct configurations to reflect realistic deployment scenarios. In the MVBA evaluation, we set $L \approx n/12$ Bytes (specifically $L = (n-f)/8$). This choice aligns with the internal usage of MVBA within the FIN-ACS construction [36], demonstrating that our protocol serves as a more efficient drop-in replacement. In the ACS evaluation, we fix $L = 32$ Bytes, a standard payload size for core applications such as Asynchronous Anonymous Broadcast [51].

[14] While FIN-MVBA/ACS have multiple variants, we analyze the most practical variants implemented in [36].

Performance Results. Figure 3 illustrates the communication cost comparison for MVBA. The results confirm our theoretical analysis: while FIN-MVBA exhibits competitive performance at very small network sizes, our protocol demonstrates superior scalability as the network grows. Specifically:

- In the 128-bit security setting, our MVBA outperforms FIN-MVBA when the network size exceeds $n \approx 105$.
- In the 256-bit security setting, the crossover point occurs at approximately $n \approx 364$.

Beyond these thresholds, the cost of FIN-MVBA grows rapidly due to its higher asymptotic complexity ($O(n^3)$), whereas our protocol maintains a more efficient growth rate. For large networks (e.g., $n = 1000$), our MVBA reduces communication overhead by roughly an order of magnitude.

Figure 4 compares the communication cost of our ACS protocol against FIN-ACS. We observe two distinct trends based on the signature instantiation:

- **Hash-based Signatures (HS):** Due to the large size of hash-based signatures, our protocol starts with a higher initial overhead than FIN-ACS. However, the efficiency of our underlying MVBA eventually dominates as n increases. For 128-bit security, Our-ACS-HS becomes more efficient than FIN-ACS at $n \approx 1344$. For 256-bit security, the crossover occurs at $n \approx 3508$.
- **Lattice-based Signatures (LS):** When instantiated with the post-quantum Falcon scheme, our protocol exhibits a significant advantage much earlier. For 128-bit security, Our-ACS-LS outperforms FIN-ACS for $n > 196$. For 256-bit security, it becomes superior for $n > 462$.

Acknowledgment

We thank the anonymous reviewers for valuable comments. This work is supported in part by ARC Discovery Project(DP250101739), and research awards from the Stellar Development Foundation, and Input Output as part of the Cardano Vision 2026 research initiative. The work of Zhenliang was supported in part by a grant from City University of Hong Kong (Project No. 9610761).

References

1. Abraham, I., Asharov, G., Patra, A., Stern, G.: Perfectly secure asynchronous agreement on a core set in constant expected time. IACR Cryptol. ePrint Arch. p. 1130 (2023)
2. Abraham, I., Chan, T.H., Dolev, D., Nayak, K., Pass, R., Ren, L., Shi, E.: Communication complexity of byzantine agreement, revisited. In: PODC. pp. 317–326. ACM (2019)
3. Abraham, I., Malkhi, D., Spiegelman, A.: Asymptotically optimal validated asynchronous byzantine agreement. In: Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing. pp. 337–346 (2019)
4. Alhaddad, N., Das, S., Duan, S., Ren, L., Varia, M., Xiang, Z., Zhang, H.: Balanced byzantine reliable broadcast with near-optimal communication and improved computation. In: Proceedings of the 2022 ACM Symposium on Principles of Distributed Computing. pp. 399–417 (2022)
5. Asharov, G., Lindell, Y.: A full proof of the BGW protocol for perfectly secure multiparty computation. J. Cryptol. **30**(1), 58–151 (2017)
6. Backes, M., Bendun, F., Choudhury, A., Kate, A.: Asynchronous mpc with a strict honest majority using non-equivocation. In: Proceedings of the 2014 ACM symposium on Principles of distributed computing. pp. 10–19 (2014)
7. Beaver, D.: Efficient multiparty protocols using circuit randomization. In: CRYPTO. Lecture Notes in Computer Science, vol. 576, pp. 420–432. Springer (1991)
8. Ben-Or, M., Canetti, R., Goldreich, O.: Asynchronous secure computation. In: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing. pp. 52–61 (1993)
9. Ben-Or, M., Kelmer, B., Rabin, T.: Asynchronous secure computations with optimal resilience (extended abstract). In: PODC. pp. 183–192. ACM (1994)
10. Bernstein, D.J., Hopwood, D., Hülsing, A., Lange, T., Niederhagen, R., Papachristodoulou, L., Schneider, M., Schwabe, P., Wilcox-O'Hearn, Z.: SPHINCS: practical stateless hash-based signatures. In: EUROCRYPT (1). Lecture Notes in Computer Science, vol. 9056, pp. 368–397. Springer (2015)

11. Blahut, R.E.: Theory and practice of error control codes. Addison-Wesley (1983)
12. Blum, E., Katz, J., Liu-Zhang, C.D., Loss, J.: Asynchronous byzantine agreement with subquadratic communication. In: Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part I 18. pp. 353–380. Springer (2020)
13. Blum, E., Liu-Zhang, C.D., Loss, J.: Always have a backup plan: fully secure synchronous mpc with asynchronous fallback. In: Annual International Cryptology Conference. pp. 707–731. Springer (2020)
14. Boneh, D., Gennaro, R., Goldfeder, S., Jain, A., Kim, S., Rasmussen, P.M.R., Sahai, A.: Threshold cryptosystems from threshold fully homomorphic encryption. In: CRYPTO (1). pp. 565–596. Lecture Notes in Computer Science, Springer (2018)
15. Bracha, G.: An asynchronous [(n-1)/3]-resilient consensus protocol. In: Proceedings of the third annual ACM symposium on Principles of distributed computing. pp. 154–162. ACM (1984)
16. Bracha, G.: Asynchronous byzantine agreement protocols. Information and Computation **75**(2), 130–143 (1987)
17. Cachin, C., Kursawe, K., Petzold, F., Shoup, V.: Secure and efficient asynchronous broadcast protocols. In: Annual International Cryptology Conference. pp. 524–541. Springer (2001)
18. Cachin, C., Mićić, J., Steinhauer, N., Zanolini, L.: Quick order fairness. In: International Conference on Financial Cryptography and Data Security. pp. 316–333. Springer (2022)
19. Cachin, C., Tessaro, S.: Asynchronous verifiable information dispersal. In: 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05). pp. 191–201. IEEE (2005)
20. Canetti, R.: Studies in secure multiparty computation and applications. Scientific Council of The Weizmann Institute of Science (1996)
21. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: FOCS. pp. 136–145. IEEE Computer Society (2001)
22. Catalano, D., Fiore, D.: Vector commitments and their applications. In: Public Key Cryptography. Lecture Notes in Computer Science, vol. 7778, pp. 55–72. Springer (2013)
23. Chopard, A., Hirt, M., Liu-Zhang, C.D.: On communication-efficient asynchronous mpc with adaptive security. In: Theory of Cryptography: 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8–11, 2021, Proceedings, Part II 19. pp. 35–65. Springer (2021)
24. Choudhury, A., Orsini, E., Patra, A., Smart, N.P.: Linear overhead optimally-resilient robust MPC using preprocessing. In: SCN. Lecture Notes in Computer Science, vol. 9841, pp. 147–168. Springer (2016)
25. Choudhury, A., Pappu, N.: Perfectly-secure asynchronous mpc for general adversaries. In: Progress in Cryptology–INDOCRYPT 2020: 21st International Conference on Cryptology in India, Bangalore, India, December 13–16, 2020, Proceedings 21. pp. 786–809. Springer (2020)
26. Choudhury, A., Patra, A.: Optimally resilient asynchronous mpc with linear communication complexity. In: Proceedings of the 16th International Conference on Distributed Computing and Networking. pp. 1–10 (2015)
27. Choudhury, A., Patra, A.: An efficient framework for unconditionally secure multiparty computation. IEEE Transactions on Information Theory **63**(1), 428–468 (2016)
28. Choudhury, A., Patra, A.: On the communication efficiency of statistically secure asynchronous MPC with optimal resilience. J. Cryptol. **36**(2), 13 (2023)
29. Cohen, R., Forghani, P., Garay, J.A., Patel, R., Zikas, V.: Concurrent asynchronous byzantine agreement in expected-constant rounds, revisited. In: TCC (4). Lecture Notes in Computer Science, vol. 14372, pp. 422–451. Springer (2023)
30. Coretti, S., Garay, J.A., Hirt, M., Zikas, V.: Constant-round asynchronous multi-party computation based on one-way functions. In: ASIACRYPT (2). Lecture Notes in Computer Science, vol. 10032, pp. 998–1021 (2016)
31. Crain, T.: Experimental evaluation of asynchronous binary byzantine consensus algorithms with $t < n/3$ and $o(n^2)$ messages and $o(1)$ round expected termination. arXiv preprint arXiv:2002.08765 (2020)
32. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. In: CRYPTO. Lecture Notes in Computer Science, vol. 4622, pp. 572–590. Springer (2007)
33. Das, S., Duan, S., Liu, S., Momose, A., Ren, L., Shoup, V.: Asynchronous consensus without trusted setup or public-key cryptography. IACR Cryptology ePrint Archive (2024), https://eprint.iacr.org/2024/677
34. Das, S., Xiang, Z., Kokoris-Kogias, L., Ren, L.: Practical asynchronous high-threshold distributed key generation and distributed polynomial sampling. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 5359–5376 (2023)
35. Das, S., Yurek, T., Xiang, Z., Miller, A., Kokoris-Kogias, L., Ren, L.: Practical asynchronous distributed key generation. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 2518–2534. IEEE (2022)
36. Duan, S., Wang, X., Zhang, H.: Fin: Practical signature-free asynchronous common subset in constant time. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 815–829 (2023)

37. Feng, H., Lu, Z., Mai, T., Tang, Q.: Faster hash-based multi-valued validated asynchronous byzantine agreement. In: DSN. pp. 303–316. IEEE (2025)
38. Feng, H., Tang, Q.: Asymptotically optimal adaptive asynchronous common coin and dkg with silent setup. In: CRYPTO. pp. 642–675 (2025)
39. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Tech. rep., Massachusetts Inst of Tech Cambridge lab for Computer Science (1982)
40. Fitzi, M., Garay, J.A.: Efficient player-optimal protocols for strong and differential consensus. In: Proceedings of the twenty-second annual symposium on Principles of distributed computing. pp. 211–220 (2003)
41. Fouque, P.A., Hoffstein, J., Kirchner, P., Lyubashevsky, V., Pornin, T., Prest, T., Ricosset, T., Seiler, G., Whyte, W., Zhang, Z., et al.: Falcon: Fast-fourier lattice-based compact signatures over ntru. Submission to the NIST's post-quantum cryptography standardization process **36**(5), 1–75 (2018)
42. Gao, Y., Lu, Y., Lu, Z., Tang, Q., Xu, J., Zhang, Z.: Dumbo-ng: Fast asynchronous bft consensus with throughput-oblivious latency. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 1187–1201 (2022)
43. Gao, Y., Lu, Y., Lu, Z., Tang, Q., Xu, J., Zhang, Z.: Efficient asynchronous byzantine agreement without private setups. In: ICDCS. pp. 246–257. IEEE (2022)
44. Guo, B., Lu, Y., Lu, Z., Tang, Q., Xu, J., Zhang, Z.: Speeding dumbo: Pushing asynchronous bft closer to practice. In: The Network and Distributed System Security Symposium (NDSS) (2022)
45. Guo, B., Lu, Z., Tang, Q., Xu, J., Zhang, Z.: Dumbo: Faster asynchronous bft protocols. In: Proc. ACM CCS 2020. ACM (2020)
46. Hu, B., Zhang, Z., Chen, H., Zhou, Y., Jiang, H., Liu, J.: DyCAPS: Asynchronous proactive secret sharing for dynamic committees. IACR Cryptology ePrint Archive (2022), https://eprint.iacr.org/2022/1169
47. Keidar, I., Kokoris-Kogias, E., Naor, O., Spiegelman, A.: All you need is dag. In: Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing. pp. 165–175 (2021)
48. Khaburzaniya, I., Chalkias, K., Lewi, K., Malvai, H.: Aggregating and thresholdizing hash-based signatures using starks. In: Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security. pp. 393–407 (2022)
49. Komatovic, J., Neu, J., Roughgarden, T.: Toward optimal-complexity hash-based asynchronous MVBA with optimal resilience. Cryptology ePrint Archive, Paper 2024/1682 (2024), https://eprint.iacr.org/2024/1682
50. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: STOC. pp. 1219–1234. ACM (2012)
51. Lu, D., Yurek, T., Kulshreshtha, S., Govind, R., Kate, A., Miller, A.: 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. pp. 887–903 (2019)
52. Lu, Y., Lu, Z., Tang, Q., Wang, G.: Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In: Proceedings of the 39th symposium on principles of distributed computing. pp. 129–138 (2020)
53. Miller, A., Xia, Y., Croman, K., Shi, E., Song, D.: The honey badger of bft protocols. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 31–42. ACM (2016)
54. Mostefaoui, A., Moumen, H., Raynal, M.: Signature-free asynchronous byzantine consensus with $t < n/3$ and $\mathcal{O}(n^2)$ messages. In: Proceedings of the 2014 ACM symposium on Principles of distributed computing. pp. 2–9. ACM (2014)
55. Mostéfaoui, A., Moumen, H., Raynal, M.: Signature-free asynchronous binary byzantine consensus with $t < n/3$, $o(n^2)$ messages, and o (1) expected time. Journal of the ACM (JACM) **62**(4), 1–21 (2015)
56. Mostéfaoui, A., Raynal, M.: Signature-free asynchronous byzantine systems: from multivalued to binary consensus with $t < n/3$, $o(n^2)$ messages, and constant time. Acta Informatica **54**, 501–520 (2017)
57. Myers, S., Sergi, M., et al.: Threshold fully homomorphic encryption and secure computation. Cryptology ePrint Archive (2011)
58. Nayak, K., Ren, L., Shi, E., Vaidya, N.H., Xiang, Z.: Improved extension protocols for byzantine broadcast and agreement. In: DISC. LIPIcs, vol. 179, pp. 28:1–28:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
59. Patra, A., Choudhury, A., Pandu Rangan, C.: Efficient asynchronous verifiable secret sharing and multiparty computation. Journal of Cryptology **28**, 49–109 (2015)
60. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. Journal of the ACM (JACM) **27**(2), 228–234 (1980)
61. Shoup, V.: A theoretical take on a practical consensus protocol. IACR Cryptol. ePrint Arch. p. 696 (2024)

62. Shoup, V., Smart, N.P.: Lightweight asynchronous verifiable secret sharing with optimal resilience. Journal of Cryptology **37**(3), 27 (2024)
63. Sui, X., Duan, S.: Signature-free atomic broadcast with optimal $O(n^2)$ messages and $O(1)$ expected time. IACR Cryptol. ePrint Arch. p. 1549 (2023)
64. Yurek, T., Xiang, Z., Xia, Y., Miller, A.: Long live the honey badger: Robust asynchronous DPSS and its applications. In: USENIX Security. pp. 5413–5430 (2023)
65. Zhang, H., Duan, S.: Pace: Fully parallelizable bft from reproposable byzantine agreement. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 3151–3164 (2022)

A Supplementary Material to Efficiency Analysis

In this section, we detail the step-by-step derivation of the communication costs for our MVBA, our ACS, FIN-MVBA, and FIN-ACS.

A.1 Concrete Cost Analysis of FIN-MVBA and FIN-ACS

Component Costs. We base our analysis on the component protocols: Weak Reliable Broadcast (WRBC) [36], Cachin-Tessaro Reliable Broadcast (CT-RBC) [19], and Reproposable Asynchronous Binary Agreement (RABA) [65].

WRC. Based on the protocol specification, the communication cost consists of three phases:

- *Proposal:* The leader sends the payload to n nodes: $n(L + C)$.
- *Echo:* Nodes exchange hashes. The cost is $n^2(H + C)$.
- *Ready:* Nodes exchange ready messages (hashes). The cost is $n^2(H + C)$.

Total cost for one WRBC instance:

$$\mathsf{Cost}_{\mathsf{WRBC}} = n(L + C) + 2n^2(H + C). \tag{1}$$

CT-RBC. The protocol proceeds in three main steps involving erasure coding:

- *Disperse:* The sender distributes coded fragments and proofs. With a fragment size of approx. $3L/n$ and proof size $H \log n$, the cost is $n(C + 3L/n + H \log n) = nC + 3L + nH \log n$.
- *Agree on the hash:* Nodes exchange consistency messages. The cost is $2n^2(C + H)$.
- *Reconstruct:* Nodes retrieve fragments to reconstruct the input. The cost is $n^2(C + 3L/n + H \log n) \approx n^2(C + H \log n) + 3nL$.

Summing these:

$$\mathsf{Cost}_{\mathsf{CT\text{-}RBC}} = n(C + H \log n + 3L) + n^2(3C + 2H + H \log n) + 3L. \tag{2}$$

RABA. We analyze the cost per round. Each round involves a common coin and three phases of message exchange, each costing n^2C.

- *Per-round cost:* $n^2C(\text{Coin}) + 3 \times n^2C = 4n^2C$.

Since RABA terminates in 4 rounds in expectation:

$$\mathsf{Cost}_{\mathsf{RABA}} = 4 \times (4n^2C) = 16n^2C. \tag{3}$$

FIN-MVBA Derivation. The FIN-MVBA protocol involves n parallel WRBC instances, followed by a sequence of leader election and RABA instances.

- *Broadcast Phase:* n instances of WRBC.

$$\text{Cost} = n \times \mathsf{Cost}_{\mathsf{WRBC}} = n^2(L + C) + 2n^3(H + C).$$

- *Election Phase:* This loop runs for 3 rounds in expectation. In each round, the protocol invokes a common coin (n^2C) and n instances of RABA.

$$\text{Cost per round} = n^2C + n \times (16n^2C) = n^2C + 16n^3C.$$

$$\text{Total Cost} = 3 \times (n^2C + 16n^3C) = 3n^2C + 48n^3C.$$

- *Final Step:* An extra multicast of the decision.

$$\text{Cost} = n^2(L + C).$$

Summing these components:

$$\begin{aligned}\mathsf{Cost}_{\mathsf{FIN\text{-}MVBA}} &= [n^2(L + C) + 2n^3(H + C)] + [3n^2C + 48n^3C] + [n^2(L + C)] \\ &= 2n^2(L + C) + 3n^2C + n^3(2H + 2C + 48C) \\ &= 2n^2L + 5n^2C + n^3(2H + 50C).\end{aligned}$$

FIN-ACS Derivation. FIN-ACS runs n instances of CT-RBC followed by one instance of MVBA.

- *Data Dissemination:* n instances of CT-RBC.

$$\begin{aligned}\text{Cost} &= n \times [n(C + H\log n + 3L) + n^2(3C + 2H + H\log n) + 3L] \\ &= n^2(C + H\log n + 3L) + n^3(3C + 2H + H\log n) + 3nL.\end{aligned}$$

- *Agreement:* One instance of FIN-MVBA. The input to this MVBA is a vector of $n - f$ indices (where $f \approx n/3$). We approximate the input size $L' \approx (2n/3) \times \log n$. Substituting L' into the $\mathsf{Cost}_{\mathsf{FIN\text{-}MVBA}}$ formula:

$$\begin{aligned}\text{Cost} &= 2n^2(2n/3\log n) + 5n^2C + n^3(2H + 50C) \\ &= 5n^2C + n^3(2H + 50C + \frac{4}{3}\log n).\end{aligned}$$

Summing the Dissemination and Agreement costs:

$$\begin{aligned}\mathsf{Cost}_{\mathsf{FIN\text{-}ACS}} &= [n^2(C + H\log n + 3L) + n^3(3C + 2H + H\log n) + 3nL] \\ &\quad + [5n^2C + n^3(2H + 50C + \frac{4}{3}\log n)] \\ &= n^2(6C + H\log n + 3L) + n^3(53C + 4H + H\log n + \frac{4}{3}\log n) + 3nL.\end{aligned}$$

Hence, the total protocol costs are listed below:

- $\mathsf{Cost}_{\mathsf{Our\text{-}MVBA}} = nL(9 + 3\kappa) + n^2(C(6 + 12\kappa + 16\log\kappa) + H(3\log n + \kappa\log n + 11\kappa))$.
- $\mathsf{Cost}_{\mathsf{Our\text{-}ACS}} = n^2(L + S + C)(7 + 2\kappa) + n^2(C(6 + 12\kappa + 16\log\kappa) + H(3\log n + \kappa\log n + 11\kappa)$.
- $\mathsf{Cost}_{\mathsf{FIN\text{-}MVBA}} = 2n^2L + 5n^2C + n^3(2H + 50C)$.
- $\mathsf{Cost}_{\mathsf{FIN\text{-}ACS}} = n^2(6C + H\log n + 3L) + n^3(53C + 4H + H\log n + \frac{4}{3}\log n) + 3nL$.

A.2 Concrete Cost Analysis of Our Protocols

Component Costs. The analysis builds upon the costs of our core components: SMID (Disperse and Recast), SMB, ARC, and the underlying Asynchronous Binary Agreement (ABA).

SMID *Disperse.* The Disperse phase involves three steps of message exchange to distribute the erasure-coded payload.

- *Distribution:* Each node sends fragments and proofs. Cost: $n^2(C + H + 3L/n + H \log n) \approx n^2(C + H \log n) + 3nL$.
- *Echo messages:* Two rounds of all-to-all control messages. Cost: $2 \times n^2 C$.

Total cost:

$$\mathsf{Cost}_{\mathsf{SMID\text{-}Disperse}} = n^2(3C + H \log n) + 3nL. \tag{4}$$

SMID *Recast.* The Recast operation allows nodes to reconstruct a value. It involves multicasting fragments and validity proofs.

- Cost approximated as $n^2(C + 3L/n + H \log n) = n^2(C + H \log n) + 3nL$.

Total cost:

$$\mathsf{Cost}_{\mathsf{SMID\text{-}Recast}} = n^2(C + H \log n) + 3nL. \tag{5}$$

Final Recast Phase. In the final step of MVBA, the decided value is reconstructed.

$$\mathsf{Cost}_{\mathrm{Final\text{-}Recast}} = 2 \times [n^2(C + 3L/n + H \log n)] = n^2(2C + 2H \log n) + 6nL. \tag{6}$$

SMB. This component reaches agreement on a value. In the worst case, it involves 7 message exchanges, where each message carries a hash and control info.

$$\mathsf{Cost}_{\mathsf{SMB}} = 7 \times n^2(C + H) = n^2(7C + 7H). \tag{7}$$

ARC. This lighter agreement primitive involves 2 rounds of exchanges.

$$\mathsf{Cost}_{\mathsf{ARC}} = 2 \times n^2(C + H) = n^2(2C + 2H). \tag{8}$$

ABA. We use the efficient ABA from [65], which has an expected cost of:

$$\mathsf{Cost}_{\mathsf{ABA}} = 16n^2 C. \tag{9}$$

Our MVBA Derivation. Our MVBA protocol integrates these components. The total cost is the sum of:

- 1 instance of SMID Disperse.
- 1 Leader Election (Common Coin): $n^2 C$.
- κ instances of SMID Recast.
- κ instances of SMB.
- 2κ instances of ARC.
- $\log \kappa$ instances of ABA.
- 1 Final Recast phase.

Summing the terms:

$$\begin{aligned}
\mathsf{Cost}_{\mathsf{Our\text{-}MVBA}} = {} & [n^2(3C + H \log n) + 3nL] \quad \text{(Disperse)} \\
& + n^2 C \quad \text{(Election)} \\
& + \kappa[n^2(C + H \log n) + 3nL] \quad \text{(Recast)} \\
& + \kappa[n^2(7C + 7H)] \quad (\mathsf{SMB}) \\
& + 2\kappa[n^2(2C + 2H)] \quad (\mathsf{ARC}) \\
& + \log \kappa[16n^2 C] \quad (\mathsf{ABA}) \\
& + [n^2(2C + 2H \log n) + 6nL] \quad \text{(Final Recast)}
\end{aligned}$$

Grouping by variable type (L, C, H):

- **L terms:** $3nL + 3\kappa nL + 6nL = nL(9 + 3\kappa)$.

- **C terms:** $n^2[3C + C + \kappa C + 7\kappa C + 4\kappa C + 16C\log\kappa + 2C] = n^2C(6 + 12\kappa + 16\log\kappa)$.
- **H terms:** $n^2[H\log n + \kappa H\log n + 7\kappa H + 4\kappa H + 2H\log n] = n^2H(3\log n + \kappa\log n + 11\kappa)$.

The total expected cost is:

$$\begin{aligned}\mathsf{Cost}_{\mathsf{Our\text{-}MVBA}} &= nL(9 + 3\kappa) \\ &\quad + n^2\Big[C(6 + 12\kappa + 16\log\kappa) + H(3\log n + \kappa\log n + 11\kappa)\Big].\end{aligned} \tag{10}$$

Our ACS Derivation. In our ACS, every node first multicasts its signed input. Then, nodes run MVBA on a vector of $(n - f)$ signed inputs.

- *Multicast Phase:* n nodes multicast a value of size $(L + S + C)$.

$$\text{Cost} = n \times n(L + S + C) = n^2(L + S + C).$$

- *MVBA Phase:* The input payload L_{mvba} consists of a vector of signed values.

Combining the multicast cost with the MVBA cost (adjusting for the aggregate input size):

$$\begin{aligned}\mathsf{Cost}_{\mathsf{Our\text{-}ACS}} &= n^2(L + S + C)(7 + 2\kappa) \\ &\quad + n^2\Big[C(6 + 12\kappa + 16\log\kappa) + H(3\log n + \kappa\log n + 11\kappa)\Big].\end{aligned} \tag{11}$$

B UC Model

In this section, we clarify our security and network model within the UC framework.

Simulation-based security. We analyze our protocols using the standard simulation-based security, which aligns with the design of most cryptographic protocols. This approach allows us to comprehensively capture all security goals of a primitive by comparing it with an "ideal" version of some desirable functionality. It eliminates the need to enumerate a list of potentially overlapping properties and is particularly useful when arguing the security of a protocol that utilizes other protocols as sub-routines. The simulation-based security is formulated with a *real world* and an *ideal world*. In the real world, the protocol is executed by nodes that exchange messages among themselves according to the protocol specification, while an adversary can interfere with the protocol execution within certain rules. In the ideal world, all nodes only communicate with a trusted third party, referred to as an ideal functionality, which assists them in obtaining the desired output based on their inputs. Informally, a protocol is considered secure if whatever an adversary can do in the real world can also be achieved in the ideal world.

In the following, we provide a brief overview of the formal description of simulation-based security, which takes into account both adaptive adversaries and asynchronous networks within the Universal Composability (UC) framework.

The real world. An n-party protocol Π consists of an n-tuple of probabilistic polynomial time (PPT) interactive Turing machines (ITMs), representing the parties $\mathcal{P}_1, \ldots, \mathcal{P}_n$, respectively. Additionally, there are two ITMs representing the adversary $\mathcal{A}$ and the environment $\mathcal{Z}$. An execution of the protocol Π consists of a series of activations of these ITMs. It starts with the environment $\mathcal{Z}$ providing inputs to and collecting outputs from the nodes and the adversary. Upon receiving inputs or other messages, a node is activated and can perform local computations, write on its output tape, or send messages to other nodes.

Network Model. We consider an *asynchronous* network where nodes are pairwise connected with *authenticated* channels. To model the worst-case scenario, we let $\mathcal{A}$ be responsible for delivering messages between honest nodes. The adversary cannot omit, change, or inject these messages. However, the adversary can reorder the messages and arbitrarily delay them, although it cannot delay them indefinitely. These requirements are formalized using the *eventual-delivery secure message-transmission* (ED-SMT) ideal functionality in [30]. Throughout this paper, whenever we say a node $\mathcal{P}_i$ sends a message to another node $\mathcal{P}_j$, we implicitly mean

that $\mathcal{P}_i$ and $\mathcal{P}_j$ are invoking the ED-SMT ideal functionality involving the adversary $\mathcal{A}$. When we say a node $\mathcal{P}_i$ *multicasts* a message, we mean $\mathcal{P}_i$ sends the message to all nodes in the network.

Corruption Model. The adversary $\mathcal{A}$ is *adaptive*, which can corrupt the nodes at any time during the protocol execution. Once a node $\mathcal{P}_i$ is corrupted, $\mathcal{P}_i$ sends its entire local state to $\mathcal{A}$, and in all future activations follows the instructions from $\mathcal{A}$. Throughout this paper, we focus on optimal resilience, which means the total number of corrupted nodes is at most $f < n/3$. At any time of the execution, the nodes that remain honest so far are referred to as *so-far-honest* nodes, and the nodes that remain honest till the end of the execution are referred to as *forever-honest* nodes. Particularly, recall that the messages are delivered by $\mathcal{A}$. If $\mathcal{A}$ corrupts a node that just sent a message, $\mathcal{A}$ can choose not to deliver this message if it has not been delivered before corruption. Such an ability of $\mathcal{A}$ is also known as *after-the-fact removal* [2], which is also captured by the ED-SMT functionality. During the protocol execution, the environment $\mathcal{Z}$ can arbitrarily communicate with the adversary $\mathcal{A}$ for an arbitrary number of times. The execution is complete when all forever-honest nodes obtain their respective outputs; the outputs are then returned to $\mathcal{Z}$. We use $\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}}$ to denote the distribution ensemble corresponding to the binary output of $\mathcal{Z}$ at the end of an execution of Π with the adversary $\mathcal{A}$ (in the real world).

The ideal world. A computation in the ideal world consists of n *dummy* nodes and an ideal functionality $\mathcal{F}$ modeled as an ITM. There is also an adversary (or called a simulator) $\mathcal{S}$, which, interacting with $\mathcal{F}$ in a restricted and clearly defined way, is supposed to mimic an adversary $\mathcal{A}$ in the real world. In the beginning, the environment $\mathcal{Z}$ provides the initial inputs to the dummy nodes and $\mathcal{S}$. The dummy nodes provide inputs to $\mathcal{F}$ and wait to collect outputs. $\mathcal{Z}$ can communicate with $\mathcal{S}$ arbitrarily. At the end of execution, $\mathcal{Z}$ collects the outputs of dummy nodes and returns a binary value. There is no restriction on how an environment provides input/output requests. However, by the protocol description, an honest node will ignore other types of messages and only accept one input. In the ideal world, the ideal functionality in general only accepts one input message from a dummy party unless it has been corrupted. We use $\mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$ to denote the distribution ensemble with respect to the binary output of $\mathcal{Z}$.

Modeling Delayed Input And Output. An ideal functionality should explicitly capture how an adversary might interfere with the protocol execution and thus exclude all unspecified interference. Due to the asynchronous nature of the network, an adversary inherently possesses the ability to arbitrarily delay messages. Conversely, an asynchronous protocol (if it satisfies termination) typically needs to progress upon being provided with $n-f$ "valid-looking" messages since waiting for the last f messages can result in indefinite waiting. Translating these facts into the ideal world, an ideal functionality should expect at most $n-f$ inputs or participants (referred to as the "core set") and proceed based on them. The "core set" is somehow selected by the adversary from those who have been activated by the environment. To capture this, Cohen et al.'s [29] approach allows the adversary to specify a delay counter for each input procedure. A node will keep pinging the ideal functionality to fetch the output across its activations until it receives an output from the ideal functionality. The delay counter is decremented every time the node pings the ideal functionality; once the counter becomes 0, the corresponding input will be included in the "core set". Similarly, the output from the ideal functionality to a node may be delayed by the adversary, and we use the delay counter to capture it as well. It's important to note that the delay counter has to be encoded in unary, ensuring that the delay must be bound by the adversary's computational resources. We follow Cohen et al.'s approach and include the delay counter for both the input process and the output process in the descriptions of all asynchronous ideal functionalities in this paper. For further discussion, refer to [29].

Modeling Adaptive Corruption in The Ideal World. In the ideal world, we allow the simulator $\mathcal{S}$ to send corruption messages of the form $(\texttt{corruption}, \texttt{sid}, \mathcal{P}_i)$ to the ideal functionality $\mathcal{F}$, indicating that the node $\mathcal{P}_i$ is to be corrupted. $\mathcal{S}$ maintains a set $\mathbb{C}$ that keeps track of all corrupted nodes. When receiving such a message, $\mathcal{F}$ first checks whether the number of corrupted nodes has reached the threshold f; if not, it updates $\mathbb{C} \leftarrow \mathbb{C} \cup \{\mathcal{P}_i\}$ and allows $\mathcal{S}$ to send future messages on behalf of $\mathcal{P}_i$. In this paper, since most functionalities do not capture the secrecy of inputs, these ideal functionalities do not need to provide the internal states of a corrupted party to the simulator. In this case, for the sake of notational simplicity, we omit the corruption message from the simulator to the ideal functionality in the description of each ideal functionality.

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

$\mathcal{F}_{\mathsf{coin}}$ proceeds as follows, running with $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ and the adversary. At the first activation, verify $(\mathtt{sid}, \{\mathcal{P}_1, \ldots, \mathcal{P}_n\})$ for the session ID $\mathtt{sid}$ and the set of nodes. Initialize: $\mathtt{participated}_i = 0$ and $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{output}} = 1$ for all $i \in [n]$; $r = \perp$.

- Upon receiving $(\mathtt{delay}, \mathtt{sid}, \mathcal{P}_i, \mathtt{type}, D)$ from the adversary for any $i \in [n]$, if $\mathtt{type} \in \{\mathtt{input}, \mathtt{output}\}$ and $D \in \mathbb{Z}$ represented in unary notation, then update $\mathsf{D}_i^{\mathtt{type}} = \max\{1, \mathsf{D}_i^{\mathtt{type}} + D\}$, and provide $(\mathtt{delay\text{-}set}, \mathtt{sid})$ to the adversary.
- Upon receiving $(\mathtt{request}, \mathtt{sid})$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure** and provide $(\mathtt{request}, \mathtt{sid}, \mathcal{P}_i)$ to the adversary.
- Upon receiving $(\mathtt{fetch}, \mathtt{sid})$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure** and the **Output Release Procedure**, and provide $(\mathtt{fetch}, \mathtt{sid}, \mathcal{P}_i)$ to the adversary.

Input Submission Procedure: If $\mathtt{participated} = 0$, and $(\mathtt{request}, \mathtt{sid})$ has ever been provided by $\mathcal{P}_i$, do the following:

- Update $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{input}} - 1$;
- If $\mathsf{D}_i^{\mathtt{input}} = 0$, update $\mathtt{participated}_i = 1$.

Output Release Procedure: If $\sum_{j \in [n]} \mathtt{participated}_j \geq f + 1$, do the following:

- Update $\mathsf{D}_i^{\mathtt{output}} = \mathsf{D}_i^{\mathtt{output}} - 1$;
- if $\mathsf{D}_i^{\mathtt{output}} = 0$, then do the following. If $r = \perp$, $r \leftarrow\!\!\$\ V$, provide $(\mathtt{output}, \mathtt{sid}, r)$ to the adversary. Set $(\mathtt{output}, \mathtt{sid}, r)$ to be sent to $\mathcal{P}_i$.

Fig. 5. The common coin functionality $\mathcal{F}_{\mathsf{coin}}^V$ for the randomness domain of V

UC security. With the real world and the ideal world, we can define the UC security of a protocol. In particular, we say a protocol Π *UC-realizes* an ideal functionality $\mathcal{F}$ if, for any PPT adversary $\mathcal{A}$, there exists a PPT simulator $\mathcal{S}$ such that, for any PPT environment $\mathcal{Z}$, it holds that $\mathsf{EXEC}_{\Pi,\mathcal{A},\mathcal{Z}} \approx \mathsf{IDEAL}_{\mathcal{F},\mathcal{S},\mathcal{Z}}$.

Hybrid Model. Let $\mathcal{G}$ be an ideal functionality. In the $\mathcal{G}$-hybrid model, we can design a protocol Π_F in which the nodes have access to $\mathcal{G}$. The UC framework guarantees that all security properties of Π_F can be preserved if we replace $\mathcal{G}$ with any protocol that UC-realizes $\mathcal{G}$. The hybrid model captures the essence of modular protocol design. Formally, we have the following results from [21].

Lemma 13 ([21]). *Suppose Π_F is a protocol that UC-realizes an ideal functionality $\mathcal{F}$ in the $\mathcal{G}$-hybrid model, and Π_G is a protocol that UC-realizes $\mathcal{G}$. If Π_F^G is a protocol obtained by replacing every call to $\mathcal{G}$ in Π_F with an execution of Π_G, then Π_F^G UC-realizes $\mathcal{F}$ without access to $\mathcal{G}$.*

Asynchronous consensus in $\mathcal{F}_{\mathsf{coin}}$-hybrid model. As emphasized by the well-known FLP impossibility result [39], randomness is necessary for asynchronous Byzantine agreement. In the computational setting, an efficient MVBA protocol can easily imply an asynchronous Byzantine agreement [17], so it is also subjected to the impossibility. This approach distills coin generation as an independent problem and focuses on the consensus part. Most asynchronous consensus protocols are typically described within the $\mathcal{F}_{\mathsf{coin}}$-hybrid model, where participants have access to the common coin ideal functionality $\mathcal{F}_{\mathsf{coin}}$. This ideal functionality provides a uniformly random value to all nodes upon receiving a sufficient number of requests. We define $\mathcal{F}_{\mathsf{coin}}^V$ in Fig. 5, parameterized by V, the domain of random values, which is adapted from [28, Fig. 3 $\mathcal{F}_{\mathsf{rand}}$]. In the rest of the paper, we follow the standard approach and present our consensus protocols in $\mathcal{F}_{\mathsf{coin}}$-hybrid model.

B.1 Ideal Functionalities for Standard Primitives

Public Key Infrastructure ($\mathcal{F}_{\mathsf{PKI}}$). We introduce the ideal functionality $\mathcal{F}_{\mathsf{PKI}}^V$ (see Fig. 6) to provide support for digital signatures, as utilized in Section 7.

Asynchronous binary agreement ($\mathcal{F}_{\mathsf{ABA}}$). Asynchronous Binary Agreement (ABA) [54] stands as the fundamental primitive in asynchronous consensus. On rough terms, it enables a collection of participants, each equipped with a binary input, to eventually converge on a binary output. It ensures that the output

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

$\mathcal{F}_{\mathsf{PKI}}$ proceeds as follows, running with $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ and $\mathcal{S}$. At the first activation, verify $(\mathtt{sid}, \{\mathcal{P}_1, \ldots, \mathcal{P}_n\})$ for the session ID $\mathtt{sid}$ and the set of node. Initialize: $vk_i = \perp$ for all $i \in [n]$.

- Upon receiving $(\mathtt{register}, \mathtt{sid}, vk)$ from $\mathcal{P}_i$ (or $\mathcal{S}$ on behalf of a corrupted node), record $vk_j = vk$.
- Upon receiving $(\mathtt{retrive}, \mathtt{sid}, j)$ from $\mathcal{P}_i$ (or $\mathcal{S}$ on behalf of a corrupted node), if $vk_j \neq \perp$, return $(\mathtt{retrived}, \mathtt{sid}, j, vk_j)$.

Fig. 6. The PKI functionality $\mathcal{F}_{\mathsf{PKI}}$

is $b \in \{0, 1\}$ if all honest nodes input b. The ideal functionality $\mathcal{F}_{\mathsf{ABA}}$ is outlined in Fig. 7, which is adapted from [29, Fig 4] by focusing on the case of binary inputs. It is easy to argue that the ABA protocols from [**?**] can UC-realize $\mathcal{F}_{\mathsf{ABA}}$ in $\mathcal{F}_{\mathsf{coin}}$-hybrid model, with communication cost of $\mathcal{O}(n^2)$ bits and the expected round complexity of $\mathcal{O}(1)$.

C The Ideal Functionality and UC Security of SMID

C.1 The Ideal Functionality $\mathcal{F}_{\mathsf{SMID}}$

In this section, we formulate SMID as an ideal functionality $\mathcal{F}_{\mathsf{SMID}}$ in Figure 8. In $\mathcal{F}_{\mathsf{SMID}}$, each party $\mathcal{P}_i$ can provide its input v_i by sending $(\mathtt{disperse}, \mathtt{sid}, v_i)$ message to the ideal functionality. When the input delay counter $\mathsf{D}_i^{\mathsf{input}}$ reaches 0, $\mathcal{F}_{\mathsf{SMID}}$ records the input and updates the variable $\mathtt{dispersed}_i = 1$, indicating that $\mathcal{P}_i$'s message has been dispersed.

To check if the dispersal phase is complete, $\mathcal{P}_i$ can send $(\mathtt{fetch}, \mathtt{status}, \mathtt{sid})$ to the ideal functionality. Once both (1) the output delay counter $\mathsf{D}_i^{\mathsf{output}}$ reaches 0 and (2) the variable done_i is set to 1, the ideal functionality returns
$(\mathtt{disperse\text{-}done}, \mathtt{sid})$ to $\mathcal{P}_i$. The variable done_i is set to 1 when at least $n - f$ parties have dispersed their inputs, i.e., $\sum_{j \in [n]} \mathtt{dispersed}_j \geq n - f$. However, the adversary can also set $\mathsf{done}_i = 1$ earlier if the number of honest parties having dispersed inputs plus the number of corrupted parties is at least $n - f$, i.e., $\sum_{j \in \mathbb{H}} \mathtt{dispersed}_j + |\mathbb{C}| \geq n - f$.

After done_i is set to 1, $\mathcal{P}_i$ can request to recast $\mathcal{P}_j$'s input by sending $(\mathtt{recast}, \mathtt{sid}, j)$ to the ideal functionality. When the input delay $\mathsf{D}_{i,j}^{\mathsf{input}}$ reaches 0, the ideal functionality records the request and updates $\mathsf{requested}_{i,j} = 1$. $\mathcal{P}_i$ can then fetch the recast result by sending $(\mathtt{fetch}, \mathtt{recast\text{-}output}, \mathtt{sid}, j)$ to the ideal functionality.

The adversary can control the output v_j received by $\mathcal{P}_i$. If $\mathcal{P}_j$ is corrupted and $(\mathtt{recast\text{-}output}, \mathtt{sid}, i, v)$ has not been delivered to $\mathcal{P}_i$, the adversary can send $(\mathtt{configure\text{-}output}, \mathtt{sid}, \mathcal{P}_i, \mathcal{P}_j, v')$ to $\mathcal{F}_{\mathsf{SMID}}$. If $\mathcal{P}_j$ has not dispersed its input ($v_j = \perp$), $\mathcal{F}_{\mathsf{SMID}}$ records $a_{i,j} = (1, v')$, indicating adversarial control over the output. Even if $\mathcal{P}_j$ has already dispersed its input, the adversary can change the output for up to f honest nodes, provided the counter $\mathtt{C\text{-}counter}_j < f$. Nonetheless, at least $f + 1$ honest nodes will still receive the original input v_j during the recast phase.

Finally, $\mathcal{P}_i$ can receive the recast output v' when $\mathsf{D}_{i,j}^{\mathsf{output}}$ reaches 0 and either (1) the adversary has configured an output or (2) at least $n - f$ honest parties have also requested to recast $\mathcal{P}_j$'s input, i.e., $\sum_{i \in \mathbb{H}} \mathsf{requested}_{i,j} \geq n - f$.

$\mathcal{F}_{\mathsf{SMID}}$ captures all security properties. In $\mathcal{F}_{\mathsf{SMID}}$, every node $\mathcal{P}_i$ can receive the message $(\mathtt{disperse\text{-}done}, \mathtt{sid})$ from the ideal functionality as long as

$$\sum_{j \in [n]} \mathtt{dispersed}_j \geq n - f,$$

which can be ensured given $n - f$ honest participants. Therefore, $\mathcal{F}_{\mathsf{SMID}}$ captures the termination property.

Regarding the validity, when an honest node $\mathcal{P}_i$ receives $(\mathtt{disperse\text{-}done}, \mathtt{sid})$ from the ideal functionality, it satisfies $\sum_{j \in \mathbb{H}} \mathtt{dispersed}_j + |\mathbb{C}| \geq n - f$. Therefore, at least $f + 1$ forever honest nodes have successfully

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

$\mathcal{F}_{\mathsf{ABA}}$ proceeds as follows, running with $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ and the adversary. At the first activation, verify $(\mathtt{sid}, \{\mathcal{P}_1, \ldots, \mathcal{P}_n\})$ for the session ID $\mathtt{sid}$ and the set of node. Initialize: $\mathtt{participated}_i = 0$, $b_i = \perp$ and $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{output}} = 1$ for all $i \in [n]$; $y = \perp$.

- Upon receiving $(\mathtt{delay}, \mathtt{sid}, \mathcal{P}_i, \mathtt{type}, D)$ from the adversary for any $i \in [n]$, if $\mathtt{type} \in \{\mathtt{input}, \mathtt{output}\}$ and $D \in \mathbb{Z}$ represented in unary notation, then update $\mathsf{D}_i^{\mathtt{type}} = \max\{1, \mathsf{D}_i^{\mathtt{type}} + D\}$, and provide $(\mathtt{delay\text{-}set}, \mathtt{sid})$ to the adversary.
- Upon receiving $(\mathtt{input}, \mathtt{sid}, b')$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure** and provide $(\mathtt{input}, \mathtt{sid}, b', \mathcal{P}_i)$ to the adversary.
- Upon receiving $(\mathtt{fetch}, \mathtt{sid})$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure** and the **Output Release Procedure**, and provide $(\mathtt{fetch}, \mathtt{sid}, \mathcal{P}_i)$ to the adversary.

Input Submission Procedure: If $\mathtt{participated}_i = 0$ and $\mathcal{P}_i$ has ever provided an input, let $(\mathtt{input}, \mathtt{sid}, b')$ be the latest input provided by $\mathcal{P}_i$, and do the following:

- Update $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{input}} - 1$;
- If $\mathsf{D}_i^{\mathtt{input}} = 0$, update $\mathtt{participated}_i = 1$, and record $b_i = b'$.

Output Release Procedure: If $\sum_{j \in [n]} \mathtt{participated}_j \geq n - f$, do the following:

- Update $\mathsf{D}_i^{\mathtt{output}} = \mathsf{D}_i^{\mathtt{output}} - 1$;
- if $\mathsf{D}_i^{\mathtt{output}} = 0$, then do the following. If $y = \perp$, set $y = b$ for a binary value b such that $b = b_i$ for at least $n - 2f$ input values b_i, and provide $(\mathtt{output}, \mathtt{sid}, y)$ to the adversary. Set $(\mathtt{output}, \mathtt{sid}, y)$ to be sent to $\mathcal{P}_i$.

Fig. 7. The aynchronous binary agreement functionality $\mathcal{F}_{\mathsf{ABA}}$

dispersed their input values, i.e. $v_j \neq \perp$ for these honest P_j, Those $f + 1$ nodes can form the subset $\mathbb{I}$ required by the validity. Furthermore, as specified in $\mathcal{F}_{\mathsf{SMID}}$, for any $v_j \neq \perp$, the adversary can change the recast output of v_j for at most f honest nodes. Therefore, at least $f + 1$ forever honest nodes can obtain the original input v_j of P_j, which ensures the validity.

C.2 UC Security of SMID

Theorem 6. *Assuming the underlying hash function is collision resistant, the protocol Π_{SMID} in Algorithm 1 UC-realizes $\mathcal{F}_{\mathsf{SMID}}$, in the presence of a computationally bounded and adaptive adversary who may corrupt up to $f < n/3$ parties.*

Proof. Let $\mathcal{A}$ be an adversary in the real world. We construct a simulator $\mathcal{S}$ in the ideal world, such that no environment $\mathcal{Z}$ can distinguish whether it is interacting with the protocol Π_{SMID} and $\mathcal{A}$, or with $\mathcal{F}_{\mathsf{SMID}}$ and $\mathcal{S}$.

Here are some "general principles" $\mathcal{S}$ will follow. $\mathcal{S}$ runs a copy of $\mathcal{A}$, and plays the role of all "so-far-honest" parties (and hybrids if any) in a simulated execution of the protocol. All inputs from $\mathcal{Z}$ to $\mathcal{A}$ are forwarded to $\mathcal{A}$, and all outputs from $\mathcal{A}$ to $\mathcal{Z}$ are forwarded to $\mathcal{Z}$. Additionally, whenever $\mathcal{A}$ corrupts a party in the simulation, $\mathcal{S}$ corrupts the same party in the ideal world by interacting with $\mathcal{F}_{\mathsf{SMID}}$, and sends $\mathcal{A}$ the party's state and thereafter follows $\mathcal{A}$'s instructions for that party. Moreover, $\mathcal{S}$ adjust delay counters for both input submission and output release procedures based on the simulated execution influenced by the adversary. Typically, the simulator delays all inputs to the ideal functionality until the first so-far-honest party outputs in the simulated execution. Whenever the so-far-honest party outputs, (1) the simulator, if it has not, delivers enough inputs to the ideal functionality so that it determines the correct output value for the party. Then, (2) the output delay for the corresponding party is set to zero. This ensures that the order of outputs seen by the environment in the ideal world is indistinguishable from that in the real world.

The simulated execution begins when the ideal functionality $\mathcal{F}_{\mathsf{SMID}}$ provides the $(\mathtt{disperse}, \mathtt{sid}, \cdot, \cdot)$ messages to $\mathcal{S}$. These messages leak the input values of "so-far-honest" parties to the simulator. Based on these values, $\mathcal{S}$ simulates the execution for $\mathcal{A}$ and interacts with $\mathcal{F}_{\mathsf{SMID}}$ as follows, while following the general principles introduced above.

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

$\mathcal{F}_{\mathsf{SMID}}$ proceeds as follows, running with $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ and the adversary. At the first activation, verify $(\mathtt{sid}, \{\mathcal{P}_1, \ldots, \mathcal{P}_n\})$ for the session ID $\mathtt{sid}$ and the set of nodes. Initialize: $\mathtt{dispersed}_i = 0$, $v_i = \perp$, $\mathtt{C\text{-}counter}_i = 0$, $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{output}} = 1$, $\mathsf{done}_i = 0$, $a_{i,j} = (0, \perp)$, $\mathsf{requested}_{i,j} = 0$, and $\mathsf{D}_{i,j}^{\mathtt{input}} = \mathsf{D}_{i,j}^{\mathtt{output}} = 1$ for all $i, j \in [n]$ and $i \neq j$. Let $\mathbb{H}$ be the set of "so-far-honest" nodes, and $\mathbb{C}$ be the set of corrupted nodes.

- Upon receiving $(\mathtt{delay}, \mathtt{sid}, \mathcal{P}_i, \mathtt{type}, D)$ (or $(\mathtt{delay}, \mathtt{sid}, \mathcal{P}_i, \mathcal{P}_j, \mathtt{type}, D)$) from the adversary for $i, j \in [n]$, if $\mathtt{type} \in \{\mathtt{input}, \mathtt{output}\}$ and $D \in \mathbb{Z}$ represented in unary notation, then update $\mathsf{D}_i^{\mathtt{type}} = \max\{1, \mathsf{D}_i^{\mathtt{type}} + D\}$ (or $\mathsf{D}_{i,j}^{\mathtt{type}} = \max\{1, \mathsf{D}_{i,j}^{\mathtt{type}} + D\}$), and provide $(\mathtt{delay\text{-}set}, \mathtt{sid})$ to the adversary.
- Upon receiving $(\mathtt{disperse}, \mathtt{sid}, v' \neq \perp)$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure-I** and provide $(\mathtt{disperse}, \mathtt{sid}, v', \mathcal{P}_i)$ to the adversary.
- Upon receiving $(\mathtt{configure\text{-}done}, \mathtt{sid}, \mathcal{P}_i)$ from the adversary, if $\sum_{j \in \mathbb{H}} \mathtt{dispersed}_j + |\mathbb{C}| \geq n - f$, update $\mathsf{done}_i = 1$.
- Upon receiving $(\mathtt{configure\text{-}output}, \mathtt{sid}, \mathcal{P}_i, \mathcal{P}_j, v')$ from the adversary, do the following: If $(\mathtt{recast\text{-}output}, \mathtt{sid}, j, v)$ has been delivered to $\mathcal{P}_i$, or $\mathcal{P}_j$ is not corrupted, ignore this message. If $v_j = \perp$, then record $a_{i,j} = (1, v')$. If $v_j \neq \perp$, and $\mathtt{C\text{-}counter}_j < f$, then record $a_{i,j} = (1, v')$, and update $\mathtt{C\text{-}counter}_j = \mathtt{C\text{-}counter}_j + 1$. Otherwise, ignore this message. Here v' can be $\perp$.
- Upon receiving $(\mathtt{fetch}, \mathtt{status}, \mathtt{sid})$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure-I** and the **Output Release Procedure-I**, and provide $(\mathtt{fetch}, \mathtt{status}, \mathtt{sid}, \mathcal{P}_i)$ to the adversary.
- Upon receiving $(\mathtt{recast}, \mathtt{sid}, j)$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure-II**.
- Upon receiving $(\mathtt{fetch}, \mathtt{recast\text{-}output}, \mathtt{sid}, j)$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure-II** and the **Output Release Procedure-II**, and provide $(\mathtt{fetch}, \mathtt{recast\text{-}output}, \mathtt{sid}, j)$ to the adversary.

Input Submission Procedure-I: If $\mathtt{dispersed}_i = 0$ and $\mathcal{P}_i$ has ever provided an input, let $(\mathtt{disperse}, \mathtt{sid}, v')$ be the latest input provided by $\mathcal{P}_i$, and do the following: (1) Update $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{input}} - 1$; (2) If $\mathsf{D}_i^{\mathtt{input}} = 0$, update $\mathtt{dispersed}_i = 1$, and record $v_i = v'$.
Output Release Procedure-I: If $\sum_{j \in [n]} \mathtt{dispersed}_j \geq n - f$, update $\mathsf{done}_i = 1$; if $\mathsf{done}_i = 1$, do the following: (1) Update $\mathsf{D}_i^{\mathtt{output}} = \mathsf{D}_i^{\mathtt{output}} - 1$; (2) If $\mathsf{D}_i^{\mathtt{output}} = 0$, set $(\mathtt{disperse\text{-}done}, \mathtt{sid})$ to be sent to $\mathcal{P}_i$.
Input Submission Procedure-II: If $\mathsf{done}_j = 1$, and $(\mathtt{recast}, \mathtt{sid}, j)$ has been provided by $\mathcal{P}_i$, do the following: (1) Update $\mathsf{D}_{i,j}^{\mathtt{input}} = \mathsf{D}_{i,j}^{\mathtt{input}} - 1$; (2) If $\mathsf{D}_{i,j}^{\mathtt{input}} = 0$, update $\mathsf{requested}_{i,j}$=1.
Output Release Procedure-II: If $a_{i,j} = (1, v') \wedge v' \neq \perp$, or $\sum_{i \in \mathbb{H}} \mathsf{requested}_{i,j} \geq n - f$, do the following: (1) Update $\mathsf{D}_{i,j}^{\mathtt{output}} = \mathsf{D}_{i,j}^{\mathtt{output}} - 1$; (2) if $\mathsf{D}_{i,j}^{\mathtt{output}} = 0$, do the following: If $a_{i,j} = (1, v')$, then set $(\mathtt{recast\text{-}output}, j, v')$ to be sent to $\mathcal{P}_i$. Otherwise, if $v_j \neq \perp$, then set $(\mathtt{recast\text{-}output}, j, v_j)$ to be sent to $\mathcal{P}_i$.

Fig. 8. The somewhat-good multi-dealer information dispersal functionality $\mathcal{F}_{\mathsf{SMID}}$

Simulating a"so-far-honest" party $\mathcal{P}_i$. Upon receiving the input of $\mathcal{P}_i$ from $\mathcal{F}_{\mathsf{SMID}}$, run the code of $\mathcal{P}_i$ with the input (as specified in Algorithm 7), until $\mathcal{P}_i$ returns the output or becomes corrupted by $\mathcal{A}$.
Further interaction with $\mathcal{F}_{\mathsf{SMID}}$. When an honest party $\mathcal{P}_i$ in the simulated execution multicasts Completed to all (i.e., executing the code in line 13), manage the delay such that $\mathtt{dispersed}_i = 1$ in the ideal world. When an honest node $\mathcal{P}_i$ returns $(\mathtt{disperse\text{-}done}, \mathsf{id})$, send $(\mathtt{configure\text{-}done}, \mathtt{sid}, \mathcal{P}_i)$ to $\mathcal{F}_{\mathsf{SMID}}$. When an honest party $\mathcal{P}_i$ returns $(\mathtt{recast\text{-}output}, s, v')$, then he sends message $(\mathtt{configure\text{-}output}, \mathtt{sid}, \mathcal{P}_s, \mathcal{P}_i, v')$ to $\mathcal{F}_{\mathsf{SMID}}$.

Note that at the point of $\mathcal{A}$'s view, the simulated execution is perfectly indistinguishable from a real execution. Since $\mathcal{S}$ just forwards the communication between $\mathcal{A}$ and $\mathcal{Z}$, it remains to show that the outputs from $\mathcal{F}_{\mathsf{SMID}}$ to honest $\mathcal{P}_i$'s are identical to those in the simulated execution (and thus distinguishable with those in a real execution).

To demonstrate this, we establish the following facts: (1) Lemma 14: When an honest node $\mathcal{P}_i$ returns $(\mathtt{disperse\text{-}done}, \mathsf{id})$, $\mathcal{S}$ can manage to set $\mathsf{done}_i = 1$ in the ideal world. Moreover, when at least $n - f$ honest parties have participated in the dispersal phase, all honest parties eventually return $(\mathtt{disperse\text{-}done}, \mathsf{id})$. (2) Lemma 15: For each "so-far-honest" party $\mathcal{P}_j$ who has multicasted the Completed message, at most f honest nodes may receive a different value other than $\mathcal{P}_j$'s input; Moreover, when at least $n - f$ honest

parties have participated the recast phase for $\mathcal{P}_j$'s value and $v_j \neq \perp$, at least $n-2f$ honest parties could receive v_j. The first guarantees the honest parties' outputs in the dispersal phase of the simulated execution are identical to the outputs from $\mathcal{F}_{\mathsf{SMID}}$. The second one ensures that $\mathcal{S}$ can always configure the output of an honest party according to its output in the simulated execution, thus the honest parties' outputs in the recast phase of the simulated execution are identical to the outputs from $\mathcal{F}_{\mathsf{SMID}}$. □

Lemma 14. *When an honest node $\mathcal{P}_i$ returns $(\texttt{disperse-done}, \mathsf{id})$, $\mathcal{S}$ can manage to set $\mathsf{done}_i = 1$ in the ideal world. Moreover, when at least $n-f$ honest parties have participated in the dispersal phase, all honest parties eventually return $(\texttt{disperse-done}, \mathsf{id})$.*

Proof. Since an honest party $\mathcal{P}_i$ outputs $(\texttt{disperse-done}, \mathsf{id})$ only after receiving $n-f$ COMPLETED messages. If an honest party $\mathcal{P}_j$ has multicasted the COMPLETED message, the simulator $\mathcal{S}$ must have updated $\texttt{dispersed}_j = 1$ in $\mathcal{F}_{\mathsf{SMID}}$. Therefore, when $\mathcal{S}$ sends $(\texttt{configure-done}, \texttt{sid}, \mathcal{P}_i)$ to $\mathcal{F}_{\mathsf{SMID}}$, $\mathsf{done}_i = 1$ will be updated accordingly.

Since every honest node $\mathcal{P}_i$ has an input value v_i, following the protocol, $\mathcal{P}_i$ multicasts FRAGMENT messages to all. Upon receiving a valid FRAGMENT message from $\mathcal{P}_i$, any honest node $\mathcal{P}_j$ sends an OK back to $\mathcal{P}_i$. With at most f malicious nodes, $\mathcal{P}_i$ can receive $n-f$ OK messages from distinct nodes. Afterward, $\mathcal{P}_i$ multicasts a COMPLETED message to all. Again, since the number of malicious nodes is at most f, so at least $n-f$ honest nodes will multicast COMPLETED messages to all. Consequently, all honest nodes eventually receive at least $n-f$ COMPLETED messages from distinct nodes, leading to the completion of the dispersal phase for all honest nodes in an SMID instance. □

Lemma 15. *Given a collision-resistant hash function and an adversary with computationally bounded power, the following holds: If a "so-far-honest" party $\mathcal{P}_j$ inputs v_j and multicast the* COMPLETED *message, then at most f honest nodes might receive fragments that do not correspond to the value v_i. Furthermore, when at least $n-f$ honest parties input j partake in the recast phase, at least $n-2f$ honest parties could recast v_j.*

Proof. Following the procedure outlined in Algorithm 1, when an honest party multicasts a COMPLETED message, it implies that at least $n-2f$ honest nodes received valid FRAGMENT messages from $\mathcal{P}_i$, and these FRAGMENT messages contain the same vector commitment (vc). Assuming the underlying hash function is collision-resistant, the reconstructed message is identical to the original input message v_j. Since $\mathcal{A}$ cannot change the received value of those $n-2f$ honest nodes, at most f honest nodes may receive a different value other than $\mathcal{P}_j$'s initial input v_j due to the adaptive adversary $\mathcal{A}$. Moreover, if an honest node received $fragment[j]$ in the dispersal phase, then it will only output a message that is reconstructed from $f+1$ valid fragments, and these fragments have the same vc as $fragment[j]$. Hence, during the recast phase, all honest nodes will multicast their $fragment[j]$ if $fragment[j] \neq \perp$. Since at least $n-2f$ honest nodes received a valid fragment message from $\mathcal{P}_j$, allowing them to receive at least $n-2f > f+1$ valid fragment messages that share the same vc. This enables these honest nodes to reconstruct the same v_i. □

D Ideal Functionality and UC Security of SMB

D.1 The Ideal Functionality $\mathcal{F}_{\mathsf{SMB}}$

In this section, we define the functionality of $\mathcal{F}_{\mathsf{SMB}}$ in Fig. 9. This functionality encapsulates all relevant properties introduced earlier. In $\mathcal{F}_{\mathsf{SMB}}$, each party $\mathcal{P}_i$ can provide its input v_i by sending the message $(\texttt{input}, \texttt{sid}, v_i)$ to the ideal functionality. When the input delay counter $\mathsf{D}_i^{\mathsf{input}}$ reaches 0, $\mathcal{F}_{\mathsf{SMB}}$ records the input and updates the variable $\texttt{participated}_i = 1$, indicating that $\mathcal{P}_i$ has participated in the instance with its input v_i.

The adversary has the ability to control when and with what values an honest party $\mathcal{P}_i$ terminates. Specifically, the adversary can send the message $(\texttt{terminate}, \texttt{sid}, v' \neq \perp)$ to configure a set of three values $\hat{v}_1$, $\hat{v}_2$, and $\hat{v}_3$ such that $\mathcal{P}_i$ must output a subset of these values according to the following rules:

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

$\mathcal{F}_{\mathsf{SMB}}$ proceeds as follows, running with $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ and the adversary. At the first activation, verify $(\mathtt{sid}, \{\mathcal{P}_1, \ldots, \mathcal{P}_n\})$ for the session ID $\mathtt{sid}$ and the set of nodes. Initialize: $\mathtt{participated}_i = 0$, $v_i = \perp$, $\mathsf{subset}_i = \perp$ and $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{output}} = 1$ for all $i \in [n]$; $\mathsf{terminate} = 0$, $\mathsf{singleInx} = \perp$, and $\hat{v}_1 = \hat{v}_2 = \hat{v}_3 = \perp$. Let $\mathbb{H}$ be the set of "so-far-honest" nodes, and $\mathbb{C}$ be the set of corrupted nodes.

- Upon receiving $(\mathtt{delay}, \mathtt{sid}, \mathcal{P}_i, \mathtt{type}, D)$ from the adversary for any $i \in [n]$, if $\mathtt{type} \in \{\mathtt{input}, \mathtt{output}\}$ and $D \in \mathbb{Z}$ represented in unary notation, then update $\mathsf{D}_i^{\mathtt{type}} = \max\{1, \mathsf{D}_i^{\mathtt{type}} + D\}$, and provide $(\mathtt{delay\text{-}set}, \mathtt{sid})$ to the adversary.
- Upon receiving $(\mathtt{input}, \mathtt{sid}, v')$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure** and provide $(\mathtt{input}, \mathtt{sid}, v', \mathcal{P}_i)$ to the adversary.
- Upon receiving $(\mathtt{terminate}, \mathtt{sid}, v' \neq \perp)$ from the adversary, if $\mathsf{terminate} = 0$, update $\mathsf{terminate} = 1$; Use $\mathbb{J}_v$ to denote the set $\{j \in \mathbb{H}, v_j = v\}$ for any $v \neq \perp$.
  - if $\hat{v}_1 = \perp$, and $|\mathbb{J}_{v'}| + |\mathbb{C}| \geq n - 2f$, update $\hat{v}_1 = v'$;
  - if $\hat{v}_2 = \perp$, $v' \neq \hat{v}_1$, and there exists c_1, c_2, such that $c_1 + c_2 = |\mathbb{C}|$, $|\mathbb{J}_{\hat{v}_1}| + c_1 \geq n - 2f$, and $|\mathbb{J}_{v'}| + c_2 \geq n - 2f$, update $\hat{v}_2 = v'$;
  - if $v' \neq \hat{v}_1 \neq \hat{v}_2$, $\sum_{j \in \mathbb{H}} \mathtt{participated}_j \geq n - f$, and there is no v^* s.t. $v_j = v^*$ for at least $n - 2f$ different $j \in \mathbb{H}$, then then update $\hat{v}_3 = v'$.
- Upon receiving $(\mathtt{decide\text{-}output}, \mathtt{sid}, \mathcal{P}_i, \mathsf{subset} \subset \{1, 2, 3\})$ from the adversary, if $\mathsf{terminate} = 1$, and $\mathcal{P}_i$ has not received its output, update $\mathsf{subset}_i = \mathsf{subset}$.
- Upon receiving $(\mathtt{fetch}, \mathtt{sid})$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure** and the **Output Release Procedure**, and provide $(\mathtt{fetch}, \mathtt{sid}, \mathcal{P}_i)$ to the adversary.

Input Submission Procedure: If $\mathtt{participated}_i = 0$ and $\mathcal{P}_i$ has ever provided an input, let $(\mathtt{input}, \mathtt{sid}, v')$ be the latest input provided by $\mathcal{P}_i$, and do the following:

- Update $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{input}} - 1$;
- If $\mathsf{D}_i^{\mathtt{input}} = 0$, update $\mathtt{participated}_i = 1$, and record $v_i = v'$.

Output Release Procedure: If (1) $\mathsf{terminate} = 1$, or (2) there exists a subset $\mathbb{J} \subset \mathbb{H}$ such that $|\mathbb{J}| \geq n - 2f$, and $\exists\ v = v_j \neq \perp$ for all $j \in \mathbb{J}$, do the following:

- If (2) and $\hat{v}_1 = \perp$, update $\hat{v}_1 = v$ and $\mathsf{terminate} = 1$. Provide $(\mathtt{leakage}, \mathtt{sid}, \hat{v}_1)$ to the adversary.
- If $\hat{v}_1 \neq \perp$, do the following:
  - Update $\mathsf{D}_i^{\mathtt{output}} = \mathsf{D}_i^{\mathtt{output}} - 1$;
  - If $\mathsf{D}_i^{\mathtt{output}} = 0$, the following steps are executed:
    1. If $\mathsf{subset}_i = \{k^*\}$ and $\mathsf{singleIdx} = \perp$, update $\mathsf{singleIdx} \leftarrow k^*$.
    2. If $\mathsf{subset}_i \neq \perp$, $\hat{v}_j \neq \perp$ for all $j \in \mathsf{subset}_i$, and either $\mathsf{singleIdx} \in \mathsf{subset}_i$ or $\mathsf{singleIdx} = \perp$, the message $(\mathtt{output}, \mathtt{sid}, \{\hat{v}_j\}_{j \in \mathsf{subset}_i})$ is sent to $\mathcal{P}_i$.
    3. Otherwise, if $\mathsf{singleIdx} = \perp$, update $\mathsf{singleIdx} \leftarrow 1$. Then, the message $(\mathtt{output}, \mathtt{sid}, \hat{v}_{\mathsf{singleIdx}})$ is sent to $\mathcal{P}_i$.

Fig. 9. The synchronized multi-valued broadcast functionality $\mathcal{F}_{\mathsf{SMB}}$

- Setting $\hat{v}_1$: The adversary can set the first value $\hat{v}_1$ to v' if the sum of the number of parties that participated with v' and the number of corrupted parties is at least $n - 2f$. Formally, $|\mathbb{J}_{v'}| + |\mathbb{C}| \geq n - 2f$.
- Setting $\hat{v}_2$: The adversary can set the second value $\hat{v}_2$ (distinct from $\hat{v}_1$) to v' if it can partition the corrupted parties such that both $\hat{v}_1$ and $\hat{v}_2$ are supported by at least $n - 2f$ parties (honest or corrupted). That is, there exist c_1 and c_2 such that $c_1 + c_2 = |\mathbb{C}|, \quad |\mathbb{J}_{\hat{v}_1}| + c_1 \geq n - 2f, \quad |\mathbb{J}_{v'}| + c_2 \geq n - 2f$.
- Setting $\hat{v}_3$: The adversary can set the third value $\hat{v}_3$ to v' only when at least $n - f$ honest nodes have participated: $\sum_{j \in \mathbb{H}} \mathtt{participated}_j \geq n - f$, and there are no $n - 2f$ nodes among them with the same input value.

The adversary can also control which subset of these three values an honest node $\mathcal{P}_i$ can output. It can send the message $(\mathtt{decide\text{-}output}, \mathtt{sid}, \mathcal{P}_i, \mathsf{subset} \subset \{1, 2, 3\})$ to the ideal functionality. If $\mathcal{P}_i$ has not yet received its output, $\mathcal{F}_{\mathsf{SMB}}$ records $\mathsf{subset}_i = \mathsf{subset}$.

A special variable $\mathsf{singleInx}$ is introduced to enforce consistency. When the first honest party $\mathcal{P}_i$ is about to output and subset_i contains only one index k^*, $\mathsf{singleInx}$ is set to k^*. After $\mathsf{singleInx}$ is configured, every party $\mathcal{P}_j$ must output a subset that includes $\hat{v}_{k^*}$, ensuring that $\mathsf{singleInx} \in \mathsf{subset}_j$.

The SMB ideal functionality of $\mathcal{F}_{\mathsf{SMB}}$ captures the security properties:

- For termination, when at least $n-2f$ forever-honest nodes provide the same input v, the Output Release Procedure in $\mathcal{F}_{\mathsf{SMB}}$ is triggered by condition (2), ensuring that every node eventually receives an output.
- For obligation, the adversary has the ability to configure the third value $\hat{v}_3$ only if $\sum_{j\in\mathbb{H}} \texttt{participated}_j \geq n-f$ and no value appears at least $n-2f$ times among honest inputs. However, if $n-2f$ forever-honest nodes provide the same input v, this condition will not hold. In particular, if it did, the adversary could corrupt the remaining f so-far-honest nodes, contradicting the assumption of $n-2f$ forever-honest nodes with the same input. Thus, the adversary can set at most two values, and each node will output at most two values.
- For validity, $\mathcal{F}_{\mathsf{SMB}}$ enforces that when an honest node outputs a single value, this value is included in every node's output. Since at most two values can appear in any output set, if $n-2f$ forever-honest nodes provide the same value v, it must be among the first two assigned values, ensuring $val_i \subset val_j$ whenever $|val_i| \leq |val_j|$ for honest P_i and P_j. Moreover, in this case, v must be either $\hat{v}_1$ or $\hat{v}_2$, ensuring $v \in val_i$ for any $|val_i| = 2$.
- For justification, the adversary can configure a value $\hat{v}_i = v$ only when $|\mathbb{J}_v| + |\mathbb{C}| \geq n-2f$, meaning that v must be an honest node's input.

D.2 UC Security of SMB

Theorem 7. *The protocol Π_{SMB} in Algorithm 2 perfectly UC-realizes $\mathcal{F}_{\mathsf{SMB}}$, in the presence of any adaptive Byzantine adversary who corrupts up to $f < n/3$ parties.*

Proof. Let $\mathcal{A}$ be an adversary in the real world. We construct a simulator $\mathcal{S}$ in the ideal world, such that no environment $\mathcal{Z}$ can distinguish whether it is interacting with the protocol Π_{SMB} and $\mathcal{A}$, or with $\mathcal{F}_{\mathsf{SMB}}$ and $\mathcal{S}$.

$\mathcal{S}$ follows the "general principles" outlined in the proof of Theorem 6. The simulated execution begins when the ideal functionality $\mathcal{F}_{\mathsf{SMB}}$ provides the messages $(\texttt{input}, \texttt{sid}, \cdot, \cdot)$ to $\mathcal{S}$. These messages leak the input values of "so-far-honest" parties to the simulator. Based on these values, $\mathcal{S}$ simulates the execution for $\mathcal{A}$ and interacts with $\mathcal{F}_{\mathsf{SMB}}$ as follows, while following the general principles introduced above.
Simulating a"so-far-honest" party $\mathcal{P}_i$. Upon receiving the input of $\mathcal{P}_i$ from $\mathcal{F}_{\mathsf{SMB}}$, run the code of $\mathcal{P}_i$ with the input (as specified in Algorithm 2), until $\mathcal{P}_i$ returns the output or becomes corrupted by $\mathcal{A}$.
Futher Interaction with $\mathcal{F}_{\mathsf{SMB}}$. Initialize $\hat{v}_1 = \hat{v}_2 = \hat{v}_3 = \perp$.

- Upon receiving $(\texttt{leakage}, \texttt{sid}, v)$ from $\mathcal{F}_{\mathsf{SMB}}$, update $\hat{v}_1 = v$.
- Keep tracking of all $values_i$ (defined in Line 10-11 in Algorithm 2) of honest parties. Whenever a new value v' appears in any of these $values_i$'s, send $(\texttt{terminate}, \texttt{sid}, v')$ to $\mathcal{F}_{\mathsf{SMB}}$, and record $\hat{v}_k = v'$ for the smallest k s.t. $\hat{v}_k = \perp$. Before sending a terminate message, make sure enough inputs whose value is v' have been recorded by $\mathcal{F}_{\mathsf{SMB}}$. Before sending the third message, make sure the inputs from all "so-far-honest" parties have been recorded by $\mathcal{F}_{\mathsf{SMB}}$.
- Upon $\mathcal{P}_i$ returns val_i, it will send message $(\texttt{decide-output}, \texttt{sid}, \mathcal{P}_i, \mathsf{subset}_i)$ to $\mathcal{F}_{\mathsf{SMB}}$ such that $\{\hat{v}_k\}_{k\in\mathsf{subset}_i} = val_i$.

Note that at the point of $\mathcal{A}$'s view, the simulated execution is perfectly indistinguishable from a real execution. Since $\mathcal{S}$ just forwards the communication between $\mathcal{A}$ and $\mathcal{Z}$, it remains to show that the outputs from $\mathcal{F}_{\mathsf{SMB}}$ to honest $\mathcal{P}_i$'s are identical to those in the simulated execution (and thus distinguishable with those in a real execution).

To demonstrate this, we establish the following results. (1) Lemma 16: There are at most three different values among all $values_i$ of honest parties. (2) Lemma 18: All values sent by $\mathcal{S}$ via the termination messages can be recorded by $\mathcal{F}_{\mathsf{SMB}}$. (3) Lemma 19: If an honest party outputs, all honest parties can eventually output; All outputs are subsets of $\{\hat{v}_1, \hat{v}_2, \hat{v}_3\}$. (4) Lemma 20: If there exists an honest party that outputs a single value v^*, then v^* is included in all other honest parties' outputs. (5) Lemma 21: If at least $n-2f$ honest parties have the same input value v, the protocol always terminates.

The first two results ensure that the list $(\hat{v}_1, \hat{v}_2, \hat{v}_3)$ maintained by $\mathcal{S}$ is identical to that maintained by $\mathcal{F}_{\mathsf{SMB}}$. Results (3) and (4) ensure that the output of an honest party by $\mathcal{F}_{\mathsf{SMB}}$ is always identical to its output in the simulated execution, once terminate is set to be 1. Result (5) ensures the protocol must terminate, if $n-2f$ honest parties have provided the inputs of the same value to $\mathcal{F}_{\mathsf{SMB}}$. Combined them, we can conclude that outputs from $\mathcal{F}_{\mathsf{SMB}}$ to honest $\mathcal{P}_i$'s are always identical to those in the simulated execution, and thus complete the proof. □

Lemma 16. *In the worst case, there are at most three different values in all $values_i$ among honest parties.*

Proof. If a value v is in $values_i$ of an honest party, according to the code, this honest party received $n-f$ $(\textsc{Val}, \mathsf{id}, v)$ messages from distinct parties. This implies that at least $n-2f$ honest parties multicast $(\textsc{Val}, \mathsf{id}, v)$ to all honest parties. It follows that at least $n-3f$ honest parties have received $(\textsc{FilterEcho}, \mathsf{id}, v)$ from at least $n-f$ distinct parties, among which at least $n-2f$ messages are from honest parties. Note that each party can send at most two FilterEcho messages, and thus there are at most $2(n-f)$ FilterEcho messages from honest parties. Therefore, the number of values that can be recorded in $values_i$ is at most $\frac{2(n-f)}{n-2f}$. Recall that we consider $n \geq 3f+1$, so the number will be at most 3. □

Lemma 17. *If at least $n-2f$ honest parties have the same input value v, then there are at most two different values in all $values_i$ among honest parties.*

Proof. If at least $n-2f$ honest parties have the same input value v, then every honest party can receive $n-2f$ $(\textsc{Filter}, \mathsf{id}, v)$ messages from distinct parties, resulting in every honest party multicasting $(\textsc{FilterEcho}, \mathsf{id}, v)$ to all. Due to $n \geq 3f+1$, according to the code, every honest party can multicast at most two different FilterEcho messages. Since every honest party multicasts $(\textsc{FilterEcho}, \mathsf{id}, v)$, every honest party can receive $n-f$ $(\textsc{FilterEcho}, \mathsf{id}, v)$ messages from distinct parties; so every honest party will multicast a $(\textsc{Val}, \mathsf{id}, v)$ message to all.

Once again, if one honest party $\mathcal{P}_i$ multicasts $(\textsc{Val}, \mathsf{id}, v')$ to all and $v \neq v'$, and another honest party $\mathcal{P}_j$ multicasts $(\textsc{Val}, \mathsf{id}, v'')$ to all with $v \neq v''$, then it implies that $\mathcal{P}_i$ received $n-f$ $(\textsc{FilterEcho}, \mathsf{id}, v')$ from distinct parties, and $\mathcal{P}_j$ also received $n-f$ $(\textsc{FilterEcho}, \mathsf{id}, v'')$ from distinct parties. According to the previous analysis, every honest party can multicast at most two FilterEcho messages, and one of them must be $(\textsc{FilterEcho}, \mathsf{id}, v)$. As a result, at least one honest party will multicast $(\textsc{FilterEcho}, \mathsf{id}, v)$, $(\textsc{FilterEcho}, \mathsf{id}, v')$, and $(\textsc{FilterEcho}, \mathsf{id}, v'')$. This leads to a contradiction unless $v' = v''$. □

Lemma 18. *When $\mathcal{S}$ sends $(\texttt{terminate}, \texttt{sid}, v')$ to $\mathcal{F}_{\mathsf{SMB}}$, $\mathcal{F}_{\mathsf{SMB}}$ will record it as $\hat{v}_k$ for some $k \in [3]$.*

Proof. Recall that $\mathcal{S}$ sends at most three termination messages. For the first value v'_1 sent by $\mathcal{S}$, since it is recorded in $values_i$ by some honest party $\mathcal{P}_i$, there are at least $n-3f$ honest parties have received $(\textsc{FilterEcho}, \mathsf{id}, v'_1)$ from at least $n-f$ distinct parties. According to the code, at least $n-2f$ honest parties have received $(\textsc{Filter}, \mathsf{id}, v'_1)$ from $n-2f$ distinct parties, which include both honest parties whose input is v'_1 and corrupted parties. Therefore, when $\mathcal{S}$ sends the first termination message $(\texttt{terminate}, \texttt{sid}, v'_1)$ to $\mathcal{F}_{\mathsf{SMB}}$, it satisfies $|\mathbb{J}_{v'_1}| + |\mathbb{C}| \geq n-2f$ and thus could be recorded as $\hat{v}_1$.

Regarding the second value v'_2 sent by $\mathcal{S}$, for the same reason, at least $n-3f$ honest parties have received $(\textsc{FilterEcho}, \mathsf{id}, v'_2)$ from at least $n-2f$ honest distinct parties. Considering $n \geq 3f+1$, there is at least one honest party who has sent both $(\textsc{FilterEcho}, \mathsf{id}, v'_1)$ and $(\textsc{FilterEcho}, \mathsf{id}, v'_2)$, which follows that this party has received $f+1$ $(\textsc{Filter}, \mathsf{id}, v'_1)$ messages and $f+1$ $(\textsc{Filter}, \mathsf{id}, v'_2)$ messages from distinct parties, including honest parties whose input value is v'_1 or v'_2 and corrupted parties. Therefore, when $\mathcal{S}$ sends the second termination message $(\texttt{terminate}, \texttt{sid}, v'_2)$ to $\mathcal{F}_{\mathsf{SMB}}$, it satisfies $|\mathbb{J}_{v'_1}| + c_1 \geq n-2f$, $|\mathbb{J}_{v'_2}| + c_2 \geq n-2f$, and $c_1 + c_2 = |\mathbb{C}|$. Therefore, it could be recorded as $\hat{v}_2$.

Finally, if at least $n-2f$ honest parties have the same input value v, then every honest party can receive $n-2f$ $(\textsc{Filter}, \mathsf{id}, v)$ messages from distinct parties, resulting in every honest party multicasting $(\textsc{FilterEcho}, \mathsf{id}, v)$ to all. Due to $n \geq 3f+1$, according to the code, every honest party can multicast at most two different FilterEcho messages. Therefore, in this case, there is no third value in $values_i$ of any honest party $\mathcal{P}_i$. Thus, if $\mathcal{S}$ found the third value v'_3 and sent it to $\mathcal{F}_{\mathsf{SMB}}$, at least $n-f$ honest parties will not have $n-2f$ inputs of the same value. In this case, v'_3 can be recorded as $\hat{v}_3$. □

Lemma 19. *If an honest party output, all honest parties can eventually output; all outputs are subsets of* $\{\hat{v}_1, \hat{v}_2, \hat{v}_3\}$.

Proof. An honest party $\mathcal{P}_i$'s output is a subset of $values_i$. As we shown in Lemma 16, it must be a subset of $\{\hat{v}_1, \hat{v}_2, \hat{v}_3\}$.

We then argue that every value v in $values_i$ will eventually appear in another honest party $\mathcal{P}_j$'s $values_j$, if $\mathcal{P}_j$ has not terminated. Particularly, according to the code, $\mathcal{P}_i$ received $n - f$ $(\text{VAL}, \mathsf{id}, v)$ messages from distinct parties. This implies that at least $n - 2f$ honest parties multicast $(\text{VAL}, \mathsf{id}, v)$ to all honest parties. Subsequently, all honest parties will multicast $(\text{VAL}, \mathsf{id}, v)$, following the procedure outlined in Algorithm 2. It also indicates that all honest parties can receive $(\text{VAL}, \mathsf{id}, v)$ from $n - f$ honest parties. Consequently, v will be added to the set $values$.

Since every honest party will multicast an AUX message containing a value in its $values_i$, all other honest parties will be able to receive it. Therefore, every honest party could eventually receive at least $n - f$ AUX messages from honest parties. All values in those AUX messages will eventually be included in $values_j$ for any honest $\mathcal{P}_j$, and thus every honest $\mathcal{P}_j$ can eventually terminate. □

Lemma 20. *If there exists an honest party that outputs a single value* v^**, then* v^* *is included in all other honest parties' outputs.*

Proof. According to the algorithm description, every honest party will output a set of values that are conveyed by at least $n - f$ AUX messages. Assume that an honest $\mathcal{P}_i$ outputs accoriding the set AUX_i of $n - f$ AUX messages, while $\mathcal{P}_j$ outputs according to AUX_j. It is easy to see that AUX_i and AUX_j have at least $n - 3f$ common messages, which means their outputs val_i and val_j have at least common value. Thus, whenever one honest party outputs a single value, it is in all other honest parties' outputs. □

Lemma 21. *If at least* $n - 2f$ *honest parties have the same input value* v*, the protocol always terminates.*

Proof. If at least $n - 2f$ honest parties have the same input value v, then every honest party can receive $n - 2f$ $(\text{FILTER}, \mathsf{id}, v)$ messages from distinct parties, resulting in every honest party multicasting $(\text{FILTERECHO}, \mathsf{id}, v)$ to all. Due to $n \geq 3f + 1$, according to the code, every honest party can multicast at most two different FILTERECHO messages. Since every honest party multicasts $(\text{FILTERECHO}, \mathsf{id}, v)$, every honest party can receive $n - f$ $(\text{FILTERECHO}, \mathsf{id}, v)$ messages from distinct parties; hence, every honest party multicast a $(\text{VAL}, \mathsf{id}, v)$ message to all, resulting in all honest parties eventually adding v to the set $values$. However, it is possible that some honest parties multicast $(\text{VAL}, \mathsf{id}, v')$ messages, where $v \neq v'$. Consequently, it is also possible that $v' \in values_i$. Therefore, all honest parties will multicast a AUX message. Now consider two cases:

- **Case 1:** If some honest parties multicast $(\text{AUX}, \mathsf{id}, v)$ messages, while another set of honest parties multicast $(\text{AUX}, \mathsf{id}, v')$ messages, following the procedure outlined in Algorithm 2, then for any honest party $\mathcal{P}_i$, the set $values_i$ will eventually contain both v and v'.
- **Case 2:** If all honest parties can multicast $(\text{AUX}, \mathsf{id}, \bar{v})$ messages, where $\bar{v} = v$ or v', it is clear that all honest parties will have $values_i = \{\bar{v}\}$.

In either case, every honest party $\mathcal{P}_i$ will meet the condition in line 14 of Algorithm 2. Therefore, every honest party $\mathcal{P}_i$ will terminate and output a set val_i. □

We establish the following corollary for highlighting that the number of possible outputs is only 2 in "good case", which justifies why two ARC instances after each SMB instance is enough. The corollary is implied by Lemma 16 and Lemma 18.

Corollary 1. *If at least* $n - 2f$ *honest parties have the same input value* v*, then there are at most two different values in all* $values_i$ *among honest parties.*

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

$\mathcal{F}_{\mathsf{ARC}}$ proceeds as follows, running with $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ and the adversary. At the first activation, verify $(\mathtt{sid}, \{\mathcal{P}_1, \ldots, \mathcal{P}_n\})$ for the session ID $\mathtt{sid}$ and the set of nodes. Initialize: $\mathtt{participated}_i = 0$, $v_i = \perp$ and $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{output}} = 1$ for all $i \in [n]$; $y = \perp$.

- Upon receiving $(\mathtt{delay}, \mathtt{sid}, \mathcal{P}_i, \mathtt{type}, D)$ from the adversary for any $i \in [n]$, if $\mathtt{type} \in \{\mathtt{input}, \mathtt{output}\}$ and $D \in \mathbb{Z}$ represented in unary notation, then update $\mathsf{D}_i^{\mathtt{type}} = \max\{1, \mathsf{D}_i^{\mathtt{type}} + D\}$, and provide $(\mathtt{delay\text{-}set}, \mathtt{sid})$ to the adversary.
- Upon receiving $(\mathtt{input}, \mathtt{sid}, v')$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure** and provide $(\mathtt{input}, \mathtt{sid}, v', \mathcal{P}_i)$ to the adversary.
- Upon receiving $(\mathtt{fetch}, \mathtt{sid})$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure** and the **Output Release Procedure**, and provide $(\mathtt{fetch}, \mathtt{sid}, \mathcal{P}_i)$ to the adversary.

Input Submission Procedure: If $\mathtt{participated}_i = 0$ and $\mathcal{P}_i$ has ever provided an input, let $(\mathtt{input}, \mathtt{sid}, v')$ be the latest input provided by $\mathcal{P}_i$, and do the following:

- Update $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{input}} - 1$;
- If $\mathsf{D}_i^{\mathtt{input}} = 0$, update $\mathtt{participated}_i = 1$, and record $v_i = v'$.

Output Release Procedure: If $\exists\ \mathbb{I} \subset [n]$ such that $|\mathbb{I}| \geq n - f$ and $\exists\ v = v_i \neq \perp$ for all $i \in \mathbb{I}$, do the following:

- If $y = \perp$, set $y = v$, provide $(\mathtt{output}, \mathtt{sid}, y)$ to the adversary;
- Update $\mathsf{D}_i^{\mathtt{output}} = \mathsf{D}_i^{\mathtt{output}} - 1$;
- if $\mathsf{D}_i^{\mathtt{output}} = 0$, set $(\mathtt{output}, \mathtt{sid}, y)$ to be sent to $\mathcal{P}_i$.

Fig. 10. The aynchronous reliable consensus functionality $\mathcal{F}_{\mathsf{ARC}}$

E Ideal Functionality, Construction, and UC Security of ARC

E.1 The Ideal Functionality $\mathcal{F}_{\mathsf{ARC}}$

In this section, we formulate the ideal functionality $\mathcal{F}_{\mathsf{ARC}}$ in Fig. 10. Note that $\mathcal{F}_{\mathsf{ARC}}$ captures all properties of ARC. Particularly, due to how the functionality returns output, it ensures that as long as an honest node outputs v, all other honest nodes eventually output v, implying *totality and agreement*. If all honest nodes have the same input value, they will output that value, thereby satisfying *validity*. Additionally, the protocol guarantees that the output value must be the input of at least $n-2f$ honest nodes, fulfilling the requirement of *justification*.

E.2 The Construction of ARC

Following the definition of ARC, if the initial state of an ARC protocol does not satisfy the validity condition, there is no assurance regarding the termination of honest nodes. Consequently, our ARC protocol is deterministic, eliminating the need for reliance on randomness. In contrast to MBA, which, due to the asynchronous network, has to depend on randomness to overcome the FLP impossibility [39]. We present a construction for ARC with IT-security, and the detailed procedure for ARC can be found in Algorithm 7. Below is a detailed description of the process of the ARC protocol:

1. *Diffusion phase* (lines 1-2). All honest nodes are multicast their input v_i via a DIFFUSION message.
2. *Echo phase* (lines 3-10). For any honest node $\mathcal{P}_i$, if it receives $n - f$ DIFFUSION messages carrying the same value v from distinct nodes, then it will multicast the value v via an ECHO message. If it receives $f + 1$ ECHO messages carrying the same value v from distinct nodes and has not multicast an ECHO message, then it will multicast an ECHO message along with the value v.
3. *output phase* (lines 11-12). For any honest node $\mathcal{P}_i$, if it receives $n - f$ ECHO messages carrying the same value v from distinct nodes, then it outputs v.

Algorithm 7 The ARC protocol Π_{ARC} with identifier id for $\mathcal{P}_i$

upon receiving input v_i **do**
  multicast (DIFFUSION, id, v_i) to all
upon receiving (DIFFUSION, id, v) from node P_j for the first time **do**
  $D_v \leftarrow D_v \cup \{j\}$
  if $|D_v| = n - f$ and ECHO has not been sent yet **then**
    multicast (ECHO, id, v)
upon receiving (ECHO, id, v) from node P_j for the first time **do**
  $EH_v \leftarrow EH_v \cup \{j\}$
  if $|EH_v| = f + 1$ and ECHO has not been sent yet **then**
    multicast (ECHO, id, v)
  if $|EH_v| = n - f$ **then**
    return v

E.3 UC Security of ARC

Theorem 8. *The protocol Π_{ARC} in Algorithm 7 perfectly UC-realizes $\mathcal{F}_{\mathsf{ARC}}$, in the presence of any adaptive Byzantine adversary who corrupts up to $f < n/3$ parties.*

Proof. Let $\mathcal{A}$ be an adversary in the real world. We construct a simulator $\mathcal{S}$ in the ideal world, such that no environment $\mathcal{Z}$ can distinguish whether it is interacting with the protocol Π_{ARC} and $\mathcal{A}$, or with $\mathcal{F}_{\mathsf{ARC}}$ and $\mathcal{S}$.

$\mathcal{S}$ follows the "general principles" outlined in the proof of Theorem 6. The simulated execution begins when the ideal functionality $\mathcal{F}_{\mathsf{ARC}}$ provides the messages $(\mathtt{input}, \mathtt{sid}, \cdot, \cdot)$ to $\mathcal{S}$. These messages leak the input values of "so-far-honest" parties to the simulator. Based on these values, $\mathcal{S}$ simulates the execution for $\mathcal{A}$ and interacts with $\mathcal{F}_{\mathsf{ARC}}$ as follows, while following the general principles introduced above.

Simulating a"so-far-honest" party $\mathcal{P}_i$. Upon receiving the input of $\mathcal{P}_i$ from $\mathcal{F}_{\mathsf{ARC}}$, run the code of $\mathcal{P}_i$ with the input (as specified in Algorithm 7), until $\mathcal{P}_i$ returns the output or becomes corrupted by $\mathcal{A}$.

Further interaction with $\mathcal{F}_{\mathsf{ARC}}$: When an honest party outputs a value v, send $(\mathtt{input}, \mathtt{sid}, v)$ on behalf of all corrupted parties, and make sure that all honest parties have participated, i.e., $\mathtt{participated}_i = 1$ for all honest $\mathcal{P}_i$.

Note that at the point of $\mathcal{A}$'s view, the simulated execution is perfectly indistinguishable from a real execution. Since $\mathcal{S}$ just forwards the communication between $\mathcal{A}$ and $\mathcal{Z}$, it remains to show that the outputs from $\mathcal{F}_{\mathsf{ARC}}$ to honest $\mathcal{P}_i$'s are identical to those in the simulated execution (and thus distinguishable with those in a real execution).

To demonstrate this, we establish the following facts: (1)Lemma 22: When an honest party outputs v, $\mathcal{S}$ can make sure at least $n - f$ parties (including corrupted and honest ones) have participated with v in the ideal functionality. (2) Lemma 23: When an honest party outputs v, all other honest parties eventually output the same v. (3) Lemma 24: If there are $n - f$ "forever-honest" parties with the input v, all honest parties will output v. The first two ensure whenever an honest party outputs in a real execution, the dummy parties output the same in the ideal world. The last one ensures that whenever the parties should output in the ideal world, they also output in the real execution. □

Lemma 22. *When an honest party outputs v, $\mathcal{S}$ can make sure at least $n - f$ parties (including corrupted and honest ones) have participated with v in the ideal functionality.*

Proof. If an honest node $\mathcal{P}_i$ outputs v, according to the Algorithm 7, then $\mathcal{P}_i$ received $n - f$ (ECHO, id, v) messages from distinct nodes. Thus, at least one honest node received $n - f$ identical (DIFFUSION, id, v) messages from distinct nodes. The senders of those DIFFUSION messages include both honest parties with v as inputs or corrupted parties. Therefore, $\mathcal{S}$ will be able to make at least $n - f$ parties participate the ideal functionality with input v. □

Lemma 23. *When an honest party outputs v, all other honest parties eventually output the same v.*

Proof. First, we argue that if an honest node $\mathcal{P}_i$ received $(\text{ECHO}, \text{id}, v)$ from at least $f+1$ distinct nodes, at least one forever honest node $\mathcal{P}_i'$ has received $(\text{DIFFUSION}, \text{id}, v)$ from $n-f$ distinct nodes. Otherwise, if no forever honest node executed Line 5-6, no honest node in the system can receive $f+1$ $(\text{ECHO}, \text{id}, v)$ messages.

Next, we show that when one honest node $\mathcal{P}_i$ multicasts $(\text{ECHO}, \text{id}, v)$ and another honest node $\mathcal{P}_j$ multicasts $(\text{ECHO}, \text{id}, v')$, then $v = v'$. If an honest node $\mathcal{P}_i$ multicasts $(\text{ECHO}, \text{id}, v)$, according to the code, $\mathcal{P}_i$ has either received $(\text{DIFFUSION}, \text{id}, v)$ from $n-f$ distinct nodes or received $(\text{ECHO}, \text{id}, v)$ from at least $f+1$ distinct nodes. According to our argument above, it means at least one forever honest node has received $(\text{DIFFUSION}, \text{id}, v)$ from $n-f$ distinct nodes. It follows that there are at least $n-2f \geq f+1$ forever honest nodes having sent $(\text{DIFFUSION}, \text{id}, v)$; Therefore, as each honest node only sends one DIFFUSION message, it is impossible for another honest $\mathcal{P}_j$ to receive $(\text{DIFFUSION}, \text{id}, v')$ from $n-f$ distinct nodes if $v' \neq v$.

If another honest node $\mathcal{P}_j$ multicasts $(\text{ECHO}, \text{id}, v')$, it implies that $\mathcal{P}_j$ received $(\text{DIFFUSION}, \text{id}, v')$ from $n-f$ distinct nodes. Since there are at most f malicious nodes, at least $n-f \geq 2f+1$ honest nodes multicast $(\text{ECHO}, \text{id}, v)$ and $(\text{ECHO}, \text{id}, v')$. If $v \neq v'$, based on the assumption $n \geq 3f+1$, it implies that one honest node multicasts two different DIFFUSION messages, leading to a contradiction. Therefore, $v = v'$.

Then, when an honest node $\mathcal{P}_i$ outputs v, according to the code, $\mathcal{P}_i$ received $n-f$ identical $(\text{ECHO}, \text{id}, v)$ messages from distinct nodes. Due to the presence of at most f malicious nodes, at least $f+1$ honest nodes have multicast $(\text{ECHO}, \text{id}, v)$ messages to all. As a result, all honest nodes will receive at least $f+1$ $(\text{ECHO}, \text{id}, v)$ messages from distinct nodes, leading to all honest nodes multicasting an ECHO message to all. Given the previous analysis, since all ECHO messages carry the same values, so all honest nodes can receive $n-f$ ECHO messages carrying the same value v. Consequently, all honest nodes output v. □

Lemma 24. *If there are $n-f$ "forever-honest" parties with the input v, all honest parties will output v.*

Proof. If $n-f$ "forever-honest" parties have the same input value v, by the code, every honest node will multicast $(\text{ECHO}, \text{id}, v)$ messages to all nodes. Hence, every honest node will receive at least $n-f$ $(\text{ECHO}, \text{id}, v)$ messages, resulting in all honest nodes outputting v. □

F The Ideal Functionality, Construction, and UC Security of ACS

F.1 The Ideal Functionality $\mathcal{F}_{\mathsf{ACS}}$

In this section, we formulate ACS as an ideal functionality $\mathcal{F}_{\mathsf{ACS}}$ [15] in Fig. 11, and $\mathcal{F}_{\mathsf{ACS}}$ satisfies all the security *properties* listed below that are considered in ACS. $\mathcal{F}_{\mathsf{ACS}}$, along with the other ideal functionalities defined in this paper, captures the adversary's ability to adaptively corrupt honest nodes based on their inputs and retract their inputs. Particularly, the input value of an honest node is leaked to the adversary, and then the adversary may corrupt the node based on the input value, and subsequently, provide a new input value on behalf of this node. Following [5] and also [28], we consider the *corruption-aware* functionalities, namely, the functionality knows which node has been corrupted by the adversary, and thus allows the simulator to submit a different input value on behalf of a corrupted node. Nonetheless, as honest nodes do not have private states in ACS, $\mathcal{F}_{\mathsf{ACS}}$ does not need to explicitly handle a corruption request from the adversary, except keeping track of the set of corrupted nodes.

F.2 The Construction of ACS

In this section, we provide a construction of ACS in $\mathcal{F}_{\mathsf{MVBA}}$ and $\mathcal{F}_{PKI}$ hybrid model. We describe our ACS protocol in the $\mathcal{F}_{\mathsf{MVBA}}$ and $\mathcal{F}_{PKI}$ hybrid model in Algorithm 8.

[15] We remark that in a concurrent work Shoup [61] formalized the ideal functionality of ACS, which, however, is for *index-ACS*, a special variant introduced in [33] where the inputs are restricted to be validated indexes in $[n]$. Moreover, the modelings in [61] directly allow the adversary to specify the "core set", which, as discussed in [29], may fail to mimic a real-world execution; In contrast, ours follows Cohen et al.'s approach [29] to avoid potential simulation failures.

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

$\mathcal{F}_{\mathsf{ACS}}$ proceeds as follows, running with $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ and the adversary. At the first activation, verify $(\mathtt{sid}, \{\mathcal{P}_1, \ldots, \mathcal{P}_n\})$ for the session ID $\mathtt{sid}$ and the set of nodes. Initialize: $\mathtt{participated}_i = 0$, $v_i = \perp$ and $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{output}} = 1$ for all $i \in [n]$; a multiset $\mathsf{commonSet} = \emptyset$.

- Upon receiving $(\mathtt{delay}, \mathtt{sid}, \mathcal{P}_i, \mathtt{type}, D)$ from the adversary for any $i \in [n]$, if $\mathtt{type} \in \{\mathtt{input}, \mathtt{output}\}$ and $D \in \mathbb{Z}$ represented in unary notation, then update $\mathsf{D}_i^{\mathtt{type}} = \max\{1, \mathsf{D}_i^{\mathtt{type}} + D\}$, and provide $(\mathtt{delay\text{-}set}, \mathtt{sid})$ to the adversary.
- Upon receiving $(\mathtt{input}, \mathtt{sid}, v')$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure** and provide $(\mathtt{input}, \mathtt{sid}, v', \mathcal{P}_i)$ to the adversary.
- Upon receiving $(\mathtt{fetch}, \mathtt{sid})$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure** and the **Output Release Procedure**, and provide $(\mathtt{fetch}, \mathtt{sid}, \mathcal{P}_i)$ to the adversary.

Input Submission Procedure: If $\mathtt{participated}_i = 0$ and $\mathcal{P}_i$ has ever provided an input, let $(\mathtt{input}, \mathtt{sid}, v')$ be the latest input provided by $\mathcal{P}_i$, and do the following:

- Update $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{input}} - 1$;
- If $\mathsf{D}_i^{\mathtt{input}} = 0$, update $\mathtt{participated}_i = 1$, and record $v_i = v'$. If $|\mathsf{commonSet}| < n - f$, then update $\mathsf{commonSet} = \mathsf{commonSet} \cup \{v_i\}$.

Output Release Procedure: If $\sum_{j \in [n]} \mathtt{participated}_j \geq n - f$, do the following:

- Update $\mathsf{D}_i^{\mathtt{output}} = \mathsf{D}_i^{\mathtt{output}} - 1$;
- if $\mathsf{D}_i^{\mathtt{output}} = 0$, set $(\mathtt{output}, \mathtt{sid}, \mathsf{commonSet})$ to be sent to $\mathcal{P}_i$.

Fig. 11. The asynchronous common subset functionality $\mathcal{F}_{\mathsf{ACS}}$

Algorithm 8 ACS protocol (for each party $\mathcal{P}_i$), adapted from Fig 3 in [17]

Let $\Sigma = \{\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Vrfy}\}$ be the underlying signature scheme.
Initialize: $\mathbb{S} \leftarrow \emptyset$, $\mathsf{VK} \leftarrow \emptyset$, $\mathsf{registered} = 0$. Define $\mathsf{Predicate}$ of $\mathcal{F}_{\mathsf{MVBA}}$ as follows: $\mathsf{Predicate}(\{(j, v_j, \sigma_j)\}_{j \in \mathbb{J}}) = 1$, if and only if: (1) $\mathbb{J} \subset [n]$; (2) $|\mathbb{J}| = n - f$; and (3) $\Sigma.\mathsf{Vrfy}(vk_j, v_j, \sigma_j) = 1$ for all $j \in \mathbb{J}$.

1: **if** $\mathsf{registered} = 0$ **then** ▷ PKI-Setup
2: $(vk_i, sk_i) \leftarrow \mathsf{KeyGen}(1^\lambda)$
3: **send** $(\mathtt{register}, \mathtt{sid}, vk_i)$ to $\mathcal{F}_{\mathsf{PKI}}$, update $\mathsf{registered} = 1$
4: **send** $(\mathtt{retrive}, \mathtt{sid}, j)$ to $\mathcal{F}_{\mathsf{PKI}}$ for all $j \in [n]$
5: **upon** receiving $(\mathtt{retrieved}, \mathtt{sid}, j, vk_j)$ from $\mathcal{F}_{\mathsf{PKI}}$ **do**
6: Update $\mathsf{VK} = \mathsf{VK} \cup \{(j, vk_j)\}$
7: **upon** receiving input v_i **do**
8: Sign v_i: $\sigma_i \leftarrow \Sigma.\mathsf{Sign}(vk_i, sk_i, v_i)$
9: **multicast** $(\textsc{Diffusion}, (v_i, \sigma_i))$ to all
10: **upon** receive $(\textsc{Diffusion}, (v_j, \sigma_j))$ message from $\mathcal{P}_j$ for the first time **do**
11: **if** $\Sigma.\mathsf{Vrfy}(vk_j, v_j, \sigma_j) = 1$ and $|\mathbb{S}| < n - f$ **then**
12: Update $\mathbb{S} = \mathbb{S} \cup \{(j, v_j, \sigma_j)\}$
13: **wait** until $|\mathbb{S}| = n - f$
14: **send** $(\mathtt{input}, \mathtt{sid}, \mathbb{S})$ to $\mathcal{F}_{\mathsf{MVBA}}$
15: **wait** receiving $(\mathtt{output}, \mathtt{sid}, \mathbb{S}')$ from $\mathcal{F}_{\mathsf{MVBA}}$
16: Parse $\mathbb{S}' = \{(j, v_j, \sigma_j)\}_{j \in \mathbb{J}}$, and let $\mathsf{commonSet} = \{(j, v_j)\}_{j \in \mathbb{J}}$.
17: **return** commonSet.

F.3 UC Security of ACS

Theorem 9. *Assuming the underlying signature scheme satisfies the existential unforgeability, the protocol Π_{ACS} UC-realizes $\mathcal{F}_{\mathsf{ACS}}$ in the $(\mathcal{F}_{\mathsf{MVBA}}, \mathcal{F}_{\mathsf{PKI}})$-hybrid model, against any computationally bounded adaptive Byzantine adversary who may corrupt up to $f < n/3$ nodes.*

Algorithm 9 The Π_{Input} protocol

Preprocessing: For each $i \in [n]$, uniformaly sample r_i from the input space, and generate $(r_{i,1}, \ldots, r_{i,n}) \leftarrow \mathsf{SS}(r_i)$. Send $(r_i, (r_{j,i})_{j\in[n]})$ to each $\mathcal{P}_i$.

1: **upon** receiving input x_i **do** ▷ For any $\mathcal{P}_i$
2: Compute $\bar{x}_i = r_i + x_i$
3: **send** $(\texttt{input}, \texttt{sid}, \bar{x}_i)$ to $\mathcal{F}_{\mathsf{ACS}}$
4: **wait** receiving $(\texttt{output}, \texttt{sid}, \mathsf{commonSet})$ from $\mathcal{F}_{\mathsf{ACS}}$
5: Parse $\mathsf{commonSet} = \{(j, \bar{x}_j)\}_{j\in\mathbb{J}}$
6: Compute $x_{j,i} = \bar{x}_j - r_{j,i}$, for each $j \in \mathbb{J}$
7: **return** $(\texttt{output}, \texttt{sid}, \mathbb{J}, \{x_{j,i}\}_{j\in\mathbb{J}})$.

Proof. Let $\mathcal{A}$ be an adversary in the real world. We construct a simulator $\mathcal{S}$ in the ideal world, scuh that no polynomial-time environment $\mathcal{Z}$ can distinguish whether it is interacting with the protocol Π_{ACS} and $\mathcal{A}$, or with $\mathcal{F}_{\mathsf{ACS}}$ and $\mathcal{S}$.

$\mathcal{S}$ follows the "general principles" outlined in the proof of Theorem 6. In the simulated execution, $\mathcal{S}$ honestly executes the codes of honest nodes, $\mathcal{F}_{\mathsf{PKI}}$ and $\mathcal{F}_{\mathsf{MVBA}}$. Moreover, it interacts with $\mathcal{F}_{\mathsf{ACS}}$ as follows: When the hybrid $\mathcal{F}_{\mathsf{MVBA}}$ returns $(\texttt{output}, \texttt{sid}, \mathbb{S} = \{(j, v_j, \sigma_j)\}_{j\in\mathbb{J}})$ to an honest node, $\mathcal{S}$ identifies the subset $\mathbb{J}_H = \mathbb{J} \cap \mathbb{H}$, where $\mathbb{H}$ is the set of all "so-far-honest" nodes. Then, $\mathcal{S}$ manages the delay to make sure all $\mathcal{P}_j$ for $j \in \mathbb{J}_H$ have participated in $\mathcal{F}_{\mathsf{ACS}}$, i.e., $\texttt{participated}_j = 1$, while sending $(\texttt{input}, \texttt{sid}, v_k)$ on behalf of all corrupted $\mathcal{P}_k$ for $k \in \mathbb{J} \setminus \mathbb{J}_H$ without delays. All other inputs from $\mathcal{P}_j$ for $j \notin \mathbb{J}$ are delayed.

As $\mathcal{S}$ honestly executes the codes of honest nodes and hybrids, at the point of $\mathcal{A}$'s view, the simulated execution is identical to a real execution. Thus, it remains to show if the outputs of honest nodes from $\mathcal{F}_{\mathsf{ACS}}$ are identical to those in the simulated execution.

By the description, with $n-f$ nodes following the protocol, all honest nodes can eventually receive $n-f$ signed values and thus send the values to $\mathcal{F}_{\mathsf{MVBA}}$. By the description of $\mathcal{F}_{\mathsf{MVBA}}$, all honest nodes will receive the same output from $\mathcal{F}_{\mathsf{MVBA}}$ which satisfies the predicate, *i.e.*, it contains $n-f$ value/signature pairs created by distinct nodes. Ensured by the existential unforgeability of the underlying signature scheme, among the $n-f$ values, each signed by an honest node must be the input value of the node. According to how $\mathcal{S}$ specifies the inputs of corrupted nodes to $\mathcal{F}_{\mathsf{ACS}}$, it follows that an honest node's output in the simulated execution is identical to the output from $\mathcal{F}_{\mathsf{ACS}}$. □

G The Ideal Functionality and UC Security of INPUT

G.1 The Input Functionality $\mathcal{F}_{\mathsf{Input}}$

To formally study the input phase, we present the ideal functionality $\mathcal{F}_{\mathsf{Input}}$ of the input phase in Fig. 12. $\mathcal{F}_{\mathsf{Input}}$ is defined w.r.t. any secret sharing scheme SS. Moreover, different from other ideal functionalities of the consensus primitives considered in this paper, it captures the private input, *i.e.*, when a node sends an input to the ideal functionality, only the input length is leaked to the simulator. On the other hand, it explicitly considers adaptive corruption, as the private input should be leaked to the adversary when the node gets corrupt.

We focus on the Shamir sharing, as it is employed in almost all AMPC protocols with a liveness guarantee. Particularly, we leverage Choudhury et al.'s [24] trick of using a preshared random value r to mask the real input x, such that each node can obtain a secret share of x by combining the share of r and the masked $\bar{x}$. The masked value $\bar{x}$ will not leak any information about x, and thus we can employ $\mathcal{F}_{\mathsf{ACS}}$ and let all honest nodes agree on a set of masked values. Remark that the input protocol can be easily generalized to AMPC protocols not based on Shamir's secret sharing. For example, in the threshold fully homomorphic encryption based MPC, we can use $\mathcal{F}_{\mathsf{ACS}}$ to let all nodes agree on a set of ciphertexts such that further computation can be carried out on them. We present an input phase protocol in the $\mathcal{F}_{\mathsf{ACS}}$-hybrid model in Algorithm 9.

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

$\mathcal{F}_{\mathsf{Input}}$ proceeds as follows, running with $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ and the adversary. At the first activation, verify $(\mathtt{sid}, \{\mathcal{P}_1, \ldots, \mathcal{P}_n\})$ for the session ID $\mathtt{sid}$ and the set of nodes. Initialize: $\mathtt{participated}_i = 0$, $x_i = \perp$ and $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{output}} = 1$ for all $i \in [n]$; $x_{i,j} = \perp$, for all $i, j \in [n]$; $\mathsf{participants} = \emptyset$. Let $\mathbb{C}$ be the set of corrupted nodes.

- Upon receiving $(\mathtt{delay}, \mathtt{sid}, \mathcal{P}_i, \mathtt{type}, D)$ from the adversary for any $i \in [n]$, if $\mathtt{type} \in \{\mathtt{input}, \mathtt{output}\}$, and $D \in \mathbb{Z}$ represented in unary notation, then update $\mathsf{D}_i^{\mathtt{type}} = \max\{1, \mathsf{D}_i^{\mathtt{type}} + D\}$, and provide $(\mathtt{delay\text{-}set}, \mathtt{sid})$ to the adversary.
- Upon receiving $(\mathtt{corrupt}, \mathtt{sid}, \mathcal{P}_i)$ from the adversary, if $|\mathbb{C}| < f$, then $\mathbb{C} = \mathbb{C} \cup \{i\}$, and return $(\mathtt{corrupted}, \mathtt{sid}, \mathcal{P}_i, x)$ to the adversary, where x is the input from $\mathcal{P}_i$.
- Upon receiving $(\mathtt{input}, \mathtt{sid}, v')$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure** and provide $(\mathtt{input}, \mathtt{sid}, |v'|, \mathcal{P}_i)$ to the adversary.
- Upon receiving $(\mathtt{fetch}, \mathtt{sid})$ from $\mathcal{P}_i$ (or the adversary on behalf of a corrupted node), run the **Input Submission Procedure** and the **Output Release Procedure**, and provide $(\mathtt{fetch}, \mathtt{sid}, \mathcal{P}_i)$ to the adversary.

Input Submission Procedure: If $\mathtt{participated}_i = 0$, and $(\mathtt{input}, \mathtt{sid}, v')$ has been provided by $\mathcal{P}_i$, do the following:

- Update $\mathsf{D}_i^{\mathtt{input}} = \mathsf{D}_i^{\mathtt{input}} - 1$;
- If $\mathsf{D}_i^{\mathtt{input}} = 0$, update $\mathtt{participated}_i = 1$; If $|\mathsf{participants}| < n - f$, update $\mathsf{participants} \leftarrow \mathsf{participants} \cup \{i\}$; Record $x_i = v'$, and update $(x_{i,1}, \ldots, x_{i,n}) \leftarrow \mathsf{SS}(x_i)$.

Output Release Procedure: If $|\mathsf{participants}| = n - f$, do the following:

- Update $\mathsf{D}_i^{\mathtt{output}} = \mathsf{D}_i^{\mathtt{output}} - 1$;
- if $\mathsf{D}_i^{\mathtt{output}} = 0$, set $(\mathtt{output}, \mathtt{sid}, \mathsf{participants}, \{(x_{j,i})\}_{j \in \mathsf{participants}})$ to be sent to $\mathcal{P}_i$.

Fig. 12. The input functionality $\mathcal{F}_{\mathsf{Input}}$ w.r.t a secret sharing scheme SS

G.2 UC Security of INPUT

Theorem 10. *The protocol Π_{input} UC-realizes $\mathcal{F}_{\mathsf{Input}}$ in the $\mathcal{F}_{\mathsf{ACS}}$-hybrid model, against any adaptive Byzantine adversary who corrupts up to $n/3$ nodes.*

Proof. For any $\mathcal{A}$, we build a simulator $\mathcal{S}$, such that any environment $\mathcal{Z}$ cannot decide whether it is interacting with Π_{input} and $\mathcal{A}$, or with $\mathcal{F}_{\mathsf{Input}}$.

The simulator $\mathcal{S}$ follows the "general principles" outlined in the proof of Theorem 6. In the simulated execution, $\mathcal{S}$ honestly executes the code of $\mathcal{F}_{\mathsf{ACS}}$. In the following, we detail how $\mathcal{S}$ simulates the preprocessing phase and a "so-far-honest" node, handles adaptive corruptions, and interacts with $\mathcal{F}_{\mathsf{Input}}$.

Simulating the preprocessing phase. For every corrupted node $\mathcal{P}_i$, uniformly sample $\{r_i, \{r_{j,i}\}_{j \in [n]}\}$ and sent them to $\mathcal{P}_i$. Send $\{r_{i,j}\}_{i \in \mathbb{C}}$ to each honest node $\mathcal{P}_j$, where $\mathbb{C}$ is the set of corrupted nodes.

Simulating a "so-far-honest" node $\mathcal{P}_i$. Upon receiving $(\mathtt{input}, \mathtt{sid}, |v'|, \mathcal{P}_i)$ from $\mathcal{F}_{\mathsf{MVBA}}$, uniformly samples a value $\bar{x}$ from the domain of v', and send $(\mathtt{input}, \mathtt{sid}, \bar{x})$ to $\mathcal{F}_{\mathsf{ACS}}$ on behalf of $\mathcal{P}_i$.

Handling adaptive corruption. When an honest node $\mathcal{P}_i$ becomes corrupted, send $(\mathtt{corrupt}, \mathtt{sid}, \mathcal{P}_i)$ to $\mathcal{F}_{\mathsf{Input}}$ and obtain $(\mathtt{corrupted}, \mathtt{sid}, \mathcal{P}_i, x)$. Then, if $(\mathtt{input}, \mathtt{sid}, x)$ has been sent to $\mathcal{F}_{\mathsf{ACS}}$ on behalf of $\mathcal{P}_i$, let $r_i \leftarrow \bar{x} - x$, and generate the secret shares of r_i for honest nodes $\{r_{i,j}\}_{j \in \mathbb{H}}$ such that the shares are consistent with $\{r_{i,j}\}_{j \in \mathbb{C}}$, where $\mathbb{H}$ and $\mathbb{C}$ are the set of honest nodes and the set of corrupted nodes, respectively. Moreover, uniformly sample $\{r_{j,i}\}_{j \in \mathbb{H}}$. Finally, return $(x, r_i, \{r_{j,i}\}_{j \in [n]})$ to the adversary.

Interaction with $\mathcal{F}_{\mathsf{Input}}$. After receiving message $(\mathtt{output}, \mathtt{sid}, \mathsf{commonSet})$ from $\mathcal{F}_{\mathsf{ACS}}$, parse $\mathsf{commonSet} = \{j, \bar{x}_j\}_{j \in \mathbb{J}}$. For $\mathbb{J}_1 = \mathbb{J} \cap \mathbb{H}$ and $j \in \mathbb{J}_1$, manage the delay such that $\mathtt{participated}_j = 1$ in $\mathcal{F}_{\mathsf{Input}}$. For $\mathbb{J}_2 = \mathbb{C} \cap \mathbb{J}$, compute $x_j = \bar{x}_j - r_j$ for all $j \in \mathbb{J}_2$, and send $(\mathtt{input}, \mathtt{sid}, x_j)$ to $\mathcal{F}_{\mathsf{Input}}$ on behalf of $\mathcal{P}_j$ without any delays.

It is easy to verify the simulated execution is perfectly indistinguishable from a real execution, and the output of a dummy node from $\mathcal{F}_{\mathsf{Input}}$ is also identical to its output in a real execution. □