# Dumbo-MVBA: Optimal Multi-Valued Validated Asynchronous Byzantine Agreement, Revisited*

Yuan Lu[1], Zhenliang Lu[1,2], Qiang Tang[1,2], and Guiling Wang[1]

[1] New Jersey Institute of Technology
[2] JDD-NJIT-ISCAS Joint Blockchain Lab
{yl768, zl425, qiang, gwang}@njit.edu

**Abstract.** Multi-valued validated asynchronous Byzantine agreement (MVBA), proposed in the elegant work of Cachin et al. (CRYPTO '01), is fundamental for critical fault-tolerant services such as atomic broadcast in the asynchronous network. It was left as an open problem to asymptotically reduce the $\mathcal{O}(\ell n^2 + \lambda n^2 + n^3)$ communication (where $n$ is the number of parties, $\ell$ is the input length, and $\lambda$ is the security parameter). Recently, Abraham et al. (PODC '19) removed the $n^3$ term to partially answer the question when input is small. However, in other typical cases, e.g., building atomic broadcast through MVBA, the input length $\ell \geq \lambda n$, and thus the communication is dominated by the $\ell n^2$ term and the problem raised by Cachin et al. remains open.
We fill the gap and answer the remaining part of the above open problem. In particular, we present two MVBA protocols with $\mathcal{O}(\ell n + \lambda n^2)$ communicated bits, which is optimal when $\ell \geq \lambda n$. We also maintain other benefits including optimal resilience to tolerate up to $n/3$ adaptive Byzantine corruptions, optimal expected constant running time, and optimal $\mathcal{O}(n^2)$ messages.
At the core of our design, we propose asynchronous provable dispersal broadcast (APDB) in which each input can be split and dispersed to every party and later recovered in an efficient way. Leveraging APDB and asynchronous binary agreement, we design an optimal MVBA protocol, Dumbo-MVBA; we also present a general self-bootstrap framework Dumbo-MVBA* to reduce the communication of any existing MVBA protocols.

## 1 Introduction

Byzantine agreement (BA) was proposed by Lamport, Pease and Shostak in their seminal papers [36,27] and considered the scenario that a few spacecraft controllers input some readings from a sensor and try to decide a common output, despite some of them are faulty [41]. The original specification of BA [36] allows input to be multi-valued, for example, the sensor's reading in the domain of $\{0,1\}^\ell$. This general case is also known as multi-valued BA [40], which generalizes the particular case of binary BA where input is restricted to either 0 or 1 [27,32].

Recently, the renewed attention to multi-valued BA is gathered in the *asynchronous* setting [2,31,18,22], due to an unprecedented demand of deploying asynchronous atomic broadcast (ABC) [13] that is usually instantiated by sequentially executing multi-valued asynchronous BA instances with some fine-tuned validity [16,10].

The elegant work of Cachin et al. in 2001 [10] proposed *external validity* for multi-valued BA and defined validated asynchronous BA, from which a simple construction of ABC can be achieved. In this multi-valued validated asynchronous BA (MVBA), each party takes a value as input and decides *one* of the values as output, as long as the decided output satisfies the external validity condition. Later, MVBA was used as a core building block to implement a broad array of fault-tolerant protocols beyond ABC [25,38,9].

The first MVBA construction was given in the same paper [10] against computationally-bounded adversaries in the authenticated setting with the random oracle and setup assumptions (e.g., PKI and established threshold cryptosystems). The solution tolerates maximal Byzantine corruptions up to $f < n/3$ and attains expected $\mathcal{O}(1)$ running time and $\mathcal{O}(n^2)$ messages, but it incurs $\mathcal{O}(\ell n^2 + \lambda n^2 + n^3)$ communicated bits, which is large. Here, $n$ is the number of parties, $\ell$ represents the

* An abridged version of the paper appeared at ACM PODC '20.

bit-length of MVBA input, and $\lambda$ is the security parameter that captures the bit-length of digital signatures. As such, Cachin et al. raised the open problem of reducing the communication of MVBA protocols (and thus improve their ABC construction) [10], which is rephrased as:

*How to asymptotically improve the communication cost of the* MVBA *protocol by an $\mathcal{O}(n)$ factor?*

After nearly twenty years, in a recent breakthrough of Abraham et al. [2], the $n^3$ term in the communication complexity was removed, and they achieved optimal $\mathcal{O}(n^2)$ word communication, conditioned on each system word can encapsulate a constant number of input values and some small-size strings such as digital signatures. Their result can be directly translated to bit communication as a partial answer to the above question, when the input length $\ell$ is small (e.g., comparable to $\lambda$).

Nevertheless, both of the above MVBA constructions contain the $\ell n^2$ term in their communication complexities, which was reported in [10] as a major obstacle-ridden factor in a few typical use-cases where the input length $\ell$ is not that small. For instance, Cachin et al. [10] noticed their ABC construction requires the underlying MVBA's input length $\ell$ to be at least $\mathcal{O}(\lambda n)$, as each MVBA input is a set of $(n-f)$ digitally signed ABC inputs. In this case, the $\ell n^2$ term becomes the dominating factor. For this reason, it was even considered in [31,18] that existing MVBA is sub-optimal for constructing ABC due to the large communication. It follows that, despite the recent breakthrough of [2], the question from [10] *remains open* for the moderately large input size $\ell \geq \mathcal{O}(\lambda n)$.

### 1.1 Our contributions

. We answer the remaining part of the open question for large inputs and present the first MVBA protocols with expected $\mathcal{O}(\ell n + \lambda n^2)$ communicated bits. More precisely,

**Theorem 1.** *There exist protocols in the authenticated setting with setup assumptions and random oracle, such that it solves the* MVBA *problem [2,10] among n parties against an adaptive adversary controlling up to $f \leq \lfloor \frac{n-1}{3} \rfloor$ parties, with expected $\mathcal{O}(\ell n + \lambda n^2)$ communicated bits and expected constant running time, where $\ell$ is the input length and $\lambda$ is a cryptographic security parameter.*

**Table 1. Asymptotic performance of MVBA protocols among $n$ parties with $\ell$-bit input and $\lambda$-bit security parameter.**

| Protocols | Comm. (Bits) | Word† | Time | Msg. |
|---|---|---|---|---|
| Cachin et al. [10]‡ | $\mathcal{O}(\ell n^2 + \lambda n^2 + n^3)$ | $\mathcal{O}(n^3)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n^2)$ |
| Abraham et al. [2] | $\mathcal{O}(\ell n^2 + \lambda n^2)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n^2)$ |
| Our Dumbo-MVBA | $\mathcal{O}(\ell n + \lambda n^2)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n^2)$ |
| Our Dumbo-MVBA⋆ | $\mathcal{O}(\ell n + \lambda n^2)$ | $\mathcal{O}(n^2)$ | $\mathcal{O}(1)$ | $\mathcal{O}(n^2)$ |

† [2] defines a word to contain a constant number of signatures/inputs.
‡ [10] realizes that their construction can be generalized against adaptive adversary, when given threshold cryptosystems with adaptive security.

Our result not only improves communication complexity upon [10,2] as illustrated in Table 1, but also is *optimal* in the asynchronous setting regarding the following performance metrics[1]:

1. The execution incurs $\mathcal{O}(\ell n + \lambda n^2)$ bits on average, which coincides with the *optimal communication* $\mathcal{O}(\ell n)$ when $\ell \geq \mathcal{O}(\lambda n)$. This optimality can be seen trivially, since each honest party has to receive the $\ell$-bit output, indicating a minimum communication of $\Omega(\ell n)$ bits.

[1] To measure communication cost, Abraham et al. [2] define a word to contain a constant number of signatures and input values, and then consider word communication. Our protocols also achieve optimal $\mathcal{O}(n^2)$ word communication as [2], because they attain (i) optimal $\mathcal{O}(n^2)$ messages and (ii) every message is not larger than a word.

2. As [2], it can tolerate an *adaptive adversary* controlling up to $\lfloor\frac{n-1}{3}\rfloor$ Byzantine parties, which achieves the *optimal resilience* in the asynchronous network according to the upper bound of resilience presented by Bracha [8].
3. Same to [10,2], it terminates in expected constant asynchronous rounds with overwhelming probability, which is essentially *asymptotically optimal* for asynchronous BA [19,3].
4. As [10,2], it attains *asymptotically optimal* $\mathcal{O}(n^2)$ messages, which meets the lower bound of the messages of optimally-resilient asynchronous BA against adaptive adversary [2,1].

### 1.2 Techniques and challenges

Let us begin with a very brief tour to revisit the existing MVBA constructions [10,2]. In the first phase of [10], each party broadcasts its input value to all others using a broadcast protocol. Once receiving sufficient values, each party informs everyone else which values it has received to form a $\mathcal{O}(n^2)$ size "matrix". Then a random party $\mathcal{P}_l$ is elected, and an asynchronous binary agreement (ABA) is run by the parties to vote on whether to output $v_l$ depending on if enough parties have already received $v_l$. The ABA will be repeated until 1 is returned. The recent study [2], instead, expands the conventional design idea of ABA and directly constructs MVBA in the following way: first, multiple rounds of broadcasts are executed by every party to form *commit* proofs. A random party $\mathcal{P}_l$ is elected. If any party already receives a *commit* proof for $v_l$, it decides to output $v_l$; and other undecided parties use $v_l$ as input to enter a repetition of the whole procedure. We can see that [2] get rid of the $\mathcal{O}(n^3)$ communication as the phase that each party receives a $\mathcal{O}(n^2)$ size matrix is removed.

We observe that in the first phase of both [10,2], every party broadcasts its own input to all parties for checking external validity, which already results in $\ell n^2$ communicated bits. Note that a MVBA protocol only outputs a single party's input, it is thus unnecessary for *every* party to send its input to all parties. Following the observation, we design Dumbo-MVBA, a novel reduction from MVBA to ABA by using a *dispersal-then-recast* methodology to reduce communication. Instead of letting each party directly send its input to everyone, we let everyone to disperse the coded *fragments* of its input across the network. Later, after the dispersal phase has been completed, the parties could (randomly) choose a dispersed value to collectively recover it. Thanks to the external predicate, all parties can locally check the validity of the recovered value, such that they can consistently decide to output the value, or to repeat random election of another dispersed value to recover.

However, challenges remain due to our multiple efficiency requirements. For example, the number of messages to disperse a value is at most linear, otherwise $n$ dispersals would cost more than quadratic messages and make MVBA not optimal regarding message complexity. The requirement rules out a few related candidates such as asynchronous verifiable information dispersal (AVID) [12,23] that needs $\mathcal{O}(n^2)$ messages to disperse a value. In addition, the protocol must terminate in expected constant time, that means at most a constant number of dispersed values will be recovered on average.

We therefore propose asynchronous provable dispersal broadcast (APDB) for the efficiency purpose, which weakens the agreement of AVID when the sender is corrupted. In this way, we realize a meaningful dispersal protocol with only $\mathcal{O}(n)$ messages. We also introduce two succinct "proofs" in APDB as hinted by the nice work of Abraham et al. [2]. During the dispersal of APDB, two proofs *lock* and *done* could be produced: (i) when any honest party delivers a *lock* proof, enough parties have delivered the coded fragments of the dispersed value, and thus the value can be collectively recovered by all honest parties, and (ii) the *done* proof attests that enough parties deliver *lock*, so all honest parties can activate ABA with input 1 and then decide 1 to jointly recover the dispersed value. To take the most advantage of APDB, we leverage the design in [2] to let the parties exchange their *done* proofs to collectively quit all dispersals, and then borrow the idea in [10] to randomly elect a party and vote via ABA to decide whether to output the elected party's input value (if the value turns to be valid after being recovered). Intuitively, this idea reduces the communication, since (i) each fragment has only $\mathcal{O}(\ell/n)$ bits, so $n$ dispersals of $\ell$-bit input incur only $\mathcal{O}(\ell n)$ bits, (ii) the parties can reconstruct a valid value after expected constant number of ABA and recoveries. See detailed discussions in Section 5.

Finally, we present another extension MVBA protocol Dumbo-MVBA$\star$, which is a general self-bootstrap technique to reduce the communication of any existing MVBA. After applying our APDB protocol, we can use small input (i.e., the "proofs" of APDB) to invoke the underlying MVBA to pick the dispersed value to recast, thus reducing the communication of the underlying MVBA. In addition, though Dumbo-MVBA$\star$ is centering around the advanced building block of MVBA instead of the basic module of binary agreement, it can better utilize MVBA to remove the rounds generating the *done* proof in APDB, which further results in a much simpler modular design.

**The paths to** ABC. To make ABC protocols practical, most of them [22,18,31,10,16] are instantiated via the component of asynchronous common subset (ACS) [5]. ACS allows every party to take a value as input, and decides a common subset as output to include sufficient input values from distinct parties. ABC can be instantiated by simply executing ACS instances sequentially [10].

There are two main methods constructing a ACS: one is initiated by Cachin et al. [10] to reduce ACS to MVBA. The other method, initiated by Ben-Or et al. [5] and recently improved by Miller et al. in HoneyBadger BFT [31], that builds ACS using $n$ asynchronous binary agreements (ABAs) directly. During the past years, the former approach (i.e., building ACS from MVBA) was considered as sub-optimal to instantiate ABC in literature [31,18], because of the large communication complexity of existing MVBA protocols.

In a very recent work Dumbo protocols [22], a novel reduction from ACS to MVBA was proposed, such that the resulting ACS will have only a constant running time compared to that depending on number of parties $n$ in [31,18], while communication and message complexities remain the same. More importantly, this is achieved despite the seemingly large communication complexity in existing MVBA, e.g., Cachin et al MVBA was used as a blackbox. In this paper, we give another evidence that the belief that MVBA is sub-optimal in [31,18] might be too pessimistic. We directly reduce the communication complexity of MVBA protocols for a factor of $n$. These two results together show that MVBA is still be the right way to construct efficient ACS!

## 2 Further Related Work

**Validity conditions**. The asynchronous BA problem [8,5,14] was studied in diverse flavors, depending on validity conditions.

*Strong validity* [20,34] requires that if an honest party outputs $v$, then $v$ is input of some honest party. This is arguably the strongest notion of validity for multi-valued BA. The sequential execution of BA instances with strong validity gives us an ABC protocol, even in the asynchronous setting. Unfortunately, implementing strong validity is not easy. In [20], the authors even proved some disappointing bounds of strong validity in the asynchronous setting, which include: (i) the maximal number of corruptions is up to $f < n/(2^\ell+1)$, and (ii) the optimal running time is $\mathcal{O}(2^\ell)$ asynchronous rounds, where $\ell$ is the input size in bit.

*Weak validity* [26,17], only requires that if all honest parties input $v$, then every honest party outputs $v$. This is one of most widely adopted validity notions for multi-valued BA. However, it states nothing about output when the honest parties have different inputs. Weak validity is strictly weaker than strong validity [20,34], except that they coincide in binary BA [11,29,32]. Abraham et al. [2] argued: it is not clear how to achieve a simple reduction from ABC to asynchronous multi-valued BA with weak validity; in particular, the sequential execution of multi-valued BA instances with weak validity fails in the asynchronous setting, because non-default output is needed for the liveness [10] or censorship resilience [31].

*External validity* was proposed by Cachin et al. [10] to circumvent the limits of above validity notions, and it requires the decided output of honest parties to satisfy a globally known predicate. This delicately tuned notion brings a few definitional advantages: (i) compared to strong validity, it is easier to be instantiated, (ii) in contrast with weak validity, ABC is simply reducible to it. For example, Cachin et al. [10] showcased a simple reduction from ABC to MVBA (with using a notion called asynchronous common subset, i.e., ACS, as a bridge). This succinct construction sequentially executes the ACS instances, each of which allows every party to propose an input value and then solicits $n-f$ input values (from distinct parties) to output. The work also instantiates ACS due to a

reduction to MVBA by centering around a specific external validity condition, namely, input/output must be a set containing $2f+1$ valid message-signature pairs generated by distinct parties, where each signed message is an ABC input. Although their reduction is simple, the communication cost (per delivered bit) in their ABC was cubic (and is still amortizedly quadratic even if using the recent technique of batching in [31]), mainly because the communication cost of the underlying MVBA module contains a quadratic term factored by the MVBA's input length.

BA **extension protocols**. In the asynchronous setting, there exist a few nice extension protocols that can invoke Byzantine agreement only using short inputs to accommodate large multi-valued input [33,35,21]. To the best of our knowledge, all the existing extension protocols all focus on weak validity, thus their techniques cannot be directly borrowed to handle external validity. The challenge in our Dumbo-MVBA$\star$ stems from that the externally valid output can come from one corrupt party. More specifically, in those BA extensions, when all honest parties have the same input value, this value will be chosen, and we will be ensured that the dispersal-recast will finish; while in our case of MVBA, when the chosen input is from a malicious party, we have to ensure this value is recoverable (realized via the recastability in our APDB) by forcing each of them to attach an extra short proof.

# 3 Problem Formulation

## 3.1 System model

We use the standard notion [2,10] to model the system consisting of $n$ parties and an adversary in the *authenticated setting*.

**Established identities & trusted setup.** There are $n$ designated parties denoted by $\{\mathcal{P}_i\}_{i\in[n]}$, where $[n]$ is short for $\{1,\ldots,n\}$ through the paper. Moreover, we consider the trusted setup of threshold cryptosystems, namely, before the start of the protocol, each party has gotten its own secret key share and the public keys as internal states. For presentation simplicity, we consider this trusted setup for granted, while in practice it can be done via a trusted dealer or distributed key generation protocols [28,7,24].

**Adaptive Byzantine corruption.** The adversary $\mathcal{A}$ can adaptively corrupt any party at any time during the course of protocol execution, until $\mathcal{A}$ already controls $f$ parties (e.g., $3f+1=n$). If a party $\mathcal{P}_i$ was not corrupted by $\mathcal{A}$ at some stage of the protocol, it followed the protocol and kept all internal states secret against $\mathcal{A}$, and we say it is *so-far-uncorrupted*. Once a party $\mathcal{P}_i$ is corrupted by $\mathcal{A}$, it leaks all internal states to $\mathcal{A}$ and remains fully controlled by $\mathcal{A}$ to arbitrarily misbehave. By convention, the corrupted party is also called *Byzantine fault*. If and only if a party is not corrupted through the entire execution, we say it is *honest*.

**Computation model.** Following standard cryptographic practices [10,11], we let the $n$ parties and the adversary $\mathcal{A}$ to be probabilistic polynomial-time interactive Turing machines (ITMs). A party $\mathcal{P}_i$ is an ITM defined by the protocol: it is activated upon receiving an incoming message to carry out some computations, update its states, possibly generate some outgoing messages, and wait for the next activation. $\mathcal{A}$ is a probabilistic ITM that runs in polynomial time (in the number of message bits generated by honest parties). Moreover, we explicitly require the message bits generated by honest parties to be probabilistic uniformly bounded by a polynomial in the security parameter $\lambda$, which was formulated as *efficiency* in [10,2] to rule out infinite protocol executions and thus restrict the run time of the adversary through the entire protocol. Same to [10] and [2], all system parameters (e.g., $n$) are bounded by polynomials in $\lambda$.

**Asynchronous network.** Any two parties are connected via an asynchronous *reliable authenticated* point-to-point channel. When a party $\mathcal{P}_i$ attempts to send a message to another party $\mathcal{P}_j$, the adversary $\mathcal{A}$ is firstly notified about the message; then, $\mathcal{A}$ fully determines when $\mathcal{P}_j$ receives the message, but cannot drop or modify this message if both $\mathcal{P}_i$ and $\mathcal{P}_j$ are honest. The network model also allows the adaptive adversary $\mathcal{A}$ to perform "after-the-fact removal", that is, when $\mathcal{A}$ is notified about some messages sent from a *so-far-uncorrupted* party $\mathcal{P}_i$, it can delay these messages until it corrupts $\mathcal{P}_i$ to drop them.

### 3.2 Design goal: validated asynchronous BA

We review hereunder the definition of (multi-valued) validated asynchronous Byzantine agreement (MVBA) due to [10,2].

**Definition 1.** *In an* MVBA *protocol with an external* Predicate : $\{0,1\}^\ell \to \{true, false\}$*, the parties take values satisfying* Predicate *as inputs and aim to output a common value satisfying* Predicate*. The* MVBA *protocol guarantees the properties down below except with negligible probability, for any identification* id*, under the influence of any probabilistic polynomial-time adaptive adversary:*

- ***Termination.*** *If every honest party* $\mathcal{P}_i$ *is activated on identification* id*, with taking as input a value* $v_i$ *s.t.* Predicate$(v_i) = true$*, then every honest party outputs a value* $v$ *for* id*.*
- ***External-Validity.*** *If an honest party outputs a value* $v$ *for* id*, then* Predicate$(v) = true$*.*
- ***Agreement.*** *If any two honest parties output* $v$ *and* $v'$ *for* id *respectively, then* $v = v'$*.*
- ***Quality.*** *If an honest party outputs* $v$ *for* id*, the probability that* $v$ *was proposed by the adversary is at most 1/2.*

We make the following remarks about the above definition:

1. *Input length.* We focus on the general case that the input length $\ell$ can be a function in $n$. We emphasize that it captures many realistic scenarios. One remarkable example is to build ABC around MVBA as in [10] where the length of each MVBA input is at least $\mathcal{O}(\lambda n)$.
2. *External-validity* is a fine-grained validity requirement of BA. In particular, it requires the common output of the honest parties to satisfy a pre-specified global predicate function.
3. *Quality* was proposed by Abraham et al. in [2], which not only rules out trivial solutions w.r.t. some trivial predicates (e.g., output a known valid value) but also captures "fairness" to prevent the adversary from fully controlling the output.

### 3.3 Quantitative metrics

We consider the following standard quantitative metrics to characterize the performance aspects of MVBA protocols:

- **Resilience**. An MVBA protocol is said $f$-resilient, if it can tolerate an (adaptive) adversary that corrupts up to $f$ parties. If $3f+1 = n$, the MVBA protocol is said to be optimally-resilient [8]. Through the paper, we focus on the optimally-resilient MVBA against adaptive adversary.
- **Message complexity**. The message complexity measures the expected number of overall messages generated among the honest parties during the protocol execution. For the optimally-resilient MVBA against adaptive adversary, the lower bound of message complexity is expected $\Omega(n^2)$ [2,1].
- **Communication complexity**. We consider the standard notion of communication complexity to characterize the expected number of bits sent among the honest parties during the protocol. For the optimally-resilient MVBA against adaptive adversary, the lower bound of communication complexity is $\Omega(\ell n + n^2)$, where the $\ell n$ term represents a trivial lower bound that all honest parties have to deliver an externally valid value of $\ell$ bits in length [31,2,1], and the $n^2$ terms is a reflection of the lower bound of message complexity.
- **Round complexity (running time)**. We follow the standard approach due to Canetti and Rabin [14] to measure the running time of protocols by asynchronous rounds. Essentially, this measurement counts the number of messaging "rounds", when the protocol is embedded into a lock-step timing model. For asynchronous BA, the expected $\mathcal{O}(1)$ round complexity is optimal [19,3].

The above communication, message and round complexities are *probabilistically uniformly bounded* (see section 4.2) independent to the adversary as [10]. This complexity notion of messages or communications brings about the advantage that is closed under modular composition. We thus can design and analyze protocols in a modular way.

# 4 Preliminaries & Notations

## 4.1 Cryptographic abstractions

Our design uses a few cryptographic primitives/protocols.

**Erasure code scheme**. A $(k, n)$-erasure code scheme [6] consists of a tuple of two deterministic algorithms $\mathsf{Enc}$ and $\mathsf{Dec}$. The Enc algorithm maps any vector $\mathbf{v} = (v_1, \cdots, v_k)$ of $k$ data fragments into an vector $\mathbf{m} = (m_1, \cdots, m_n)$ of $n$ coded fragments, such that any $k$ elements in the code vector $\mathbf{m}$ is enough to reconstruct $\mathbf{v}$ due to the $\mathsf{Dec}$ algorithm.

More formally, a $(k, n)$-erasure code scheme has a tuple of two deterministic algorithms:

1. $\mathsf{Enc}(\mathbf{v}) \to \mathbf{m}$. On input a vector $\mathbf{v} \in \mathcal{B}^k$, this *deterministic* encode algorithm outputs a vector $\mathbf{m} \in \mathcal{B}^n$. Note that $\mathbf{v}$ contains $k$ data fragments and $\mathbf{m}$ contains $n$ coded fragments, and $\mathcal{B}$ denotes the field of each fragment.
2. $\mathsf{Dec}(\{(i, m_i)\}_{i \in S}) \to \mathbf{v}$. On input a set $\{(i, m_i)\}_{i \in S}$ where $m_i \in \mathcal{B}$, and $S \subset [n]$ and $|S| = k$, this *deterministic* decode algorithm outputs a vector of data fragments $\mathbf{v} \in \mathcal{B}^k$.

We require $(k, n)$-erasure code scheme is *maximum distance separable*, namely, the original data fragments $\mathbf{v}$ can be recovered from any $k$-size subset of the coded fragments $\mathbf{m}$, which can be formally defined as:

- **Correctness of erasure code**. For any $\mathbf{v} \in \mathcal{B}^k$ and any $S \subset [n]$ that $|S| = k$, $\Pr[\mathsf{Dec}(\{(i, m_i)\}_{i \in S}) = \mathbf{v} \mid \mathbf{m} := (m_1, \cdots, m_n) \leftarrow \mathsf{Enc}(\mathbf{v})] = 1$. If a vector $\mathbf{m} \in \mathcal{B}^n$ is indeed the coded fragments of some $\mathbf{v} \in \mathcal{B}^k$, we say the $\mathbf{m}$ is well-formed; otherwise, we say the $\mathbf{m}$ is ill-formed.

*Instantiation.* Through the paper, we consider a $(f + 1, n)$-erasure code scheme where $3f + 1 = n$. Besides, we emphasize the erasure code scheme would implicitly choose a proper $\mathcal{B}$ according to the actual length of each element in $\mathbf{v}$, such that the encoding causes only constant blow-up in size, namely, the bits of $\mathbf{m}$ are different from the bits of $\mathbf{v}$ by at most a constant factor. One well-known instantiation of such the primitive is due to Rabin [37].

**Position-binding vector commitment (VC)**. For an established position-binding $n$-vector commitment (VC), there is a tuple of algorithms ($\mathsf{VCom}, \mathsf{Open}, \mathsf{VerifyOpen}$). On input a vector $\mathbf{m}$ of any $n$ elements, the algorithm $\mathsf{VCom}$ produces a commitment $vc$ for the vector $\mathbf{m}$. On input $\mathbf{m}$ and $vc$, the $\mathsf{Open}$ algorithm can reveal the element $m_i$ committed in $vc$ at the $i$-th position while producing a short proof $\pi_i$, which later can be verified by $\mathsf{VerifyOpen}$.

Formally, a position-binding VC scheme (without hiding) is abstracted as:

1. $\mathsf{VC.Setup}(\lambda, n, \mathcal{M}) \to pp$. Given security parameter $\lambda$, the size $n$ of the input vector, and the message space $\mathcal{M}$ of each vector element, it outputs public parameters $pp$, which are implicit inputs to all the following algorithms. We explicitly require $\mathcal{M} = \{0, 1\}^*$, such that one VC scheme can commit any $n$-sized vectors.
2. $\mathsf{VCom}(\mathbf{m}) \to (vc; aux)$. On input a vector $\mathbf{m} = (m_1, ..., m_n)$, it outputs a commitment string $vc$ and an auxiliary advice string $aux$. We might omit $aux$ for presentation simplicity. Note we do not require the hiding property, and then let $\mathsf{VCom}$ to be a deterministic algorithm.
3. $\mathsf{Open}(vc, m_i, i; aux) \to \pi_i$. On input $m_i \in \mathcal{M}$, $i \in [n]$, the commitment $vc$ and advice $aux$, it produces an opening string $\pi$ to prove that $m_i$ is the $i$-th committed element. We might omit $aux$ for presentation simplicity.
4. $\mathsf{VerifyOpen}(vc, m_i, i, \pi_i) \to 0/1$. On input $m_i \in \mathcal{M}$ and $i \in [n]$, the commitment $vc$, and an opening proof $\pi$, the algorithm outputs 0 (accept) or 1 (reject).

An already established VC scheme shall satisfy **correctness** and **position binding**:

- **Correctness of VC**. An established VC scheme with public parameter $pp$ is correct, if for all $\mathbf{m} \in \mathcal{M}^n$ and $i \in [n]$, $\Pr[\mathsf{VC.VerifyOpen}(vc, m_i, i, \mathsf{VC.Open}(vc, m_i, i, aux)) = 1 \mid (vc, aux) \leftarrow \mathsf{VC.VCom}(\mathbf{m})] = 1$.

- **Position binding**. An established $\mathsf{VC}$ scheme with public parameter $pp$ is said position binding, if for any P.P.T. adversary $\mathcal{A}$, $\Pr[\mathsf{VC.VerifyOpen}(vc, m, i, \pi) = \mathsf{VC.VerifyOpen}(vc, m', i, \pi') = 1 \wedge m \neq m' \mid (vc, i, m, m', \pi, \pi') \leftarrow \mathcal{A}(pp)] < negl(\lambda)$, where $negl(\lambda)$ is a negligible function in $\lambda$.

*Instantiation.* There are a few simple solutions to achieve the above $\mathsf{VC}$ notion. An example is hash Merkle tree [30] where the commitment $vc$ is $\mathcal{O}(\lambda)$-bit and the openness $\pi$ is $\mathcal{O}(\lambda \log n)$-bit. Moreover, when given computational Diffie-Hellman assumption and collision-resistant hash function, there is a position-binding vector commitment scheme [15], s.t., all algorithms (except setup) are deterministic, and both commitment and openness are $\mathcal{O}(\lambda)$ bits.

Relying on computational Diffie-Hellman (CDH) assumption and collision-resistant hash function, there is a construction due to Catalano et al. [15] that realizes the position-binding vector commitment as follows:

- $\mathsf{VC.Setup}(\lambda, n) \rightarrow pp$. Let $\mathcal{H}$ be a collision-resistant hash function $\mathcal{H} : \{0,1\}^* \rightarrow \mathbb{Z}_p$. Given $\lambda$, generate $\mathbb{G}$ and $\mathbb{G}_T$ that are two bilinear groups of prime order $p$ with a bilinear map $e : \mathbb{G} \times \mathbb{G} \rightarrow \mathbb{G}_T$; also choose $g$ that is a random generator of $\mathbb{G}$. Randomly choose $(z_1, \cdots, z_n)$ from $\mathbb{Z}_p$, and for each $i \in [n]$, compute $h_i \leftarrow g^{z_i}$. For each $i, j \in [n]$ and $i \neq j$, compute $h_{i,j} = g^{z_i z_j}$. Set $pp = (g, \{h_i\}_{i \in [n]}, \{h_{i,j}\}_{i,j \in [n], i \neq j})$.
- $\mathsf{VCom}(\mathbf{m}) \rightarrow (vc; aux)$. Compute $vc = \prod_{i=1}^{n} h_i^{\mathcal{H}(m_i)}$ and $aux = (\mathcal{H}(m_1), \cdots, \mathcal{H}(m_n))$. The output $aux$ might be omitted in the paper for simplicity.
- $\mathsf{Open}(vc, m_i, i; aux) \rightarrow \pi_i$. Compute $\pi_i = \prod_{j=1, j \neq i}^{n} h_{i,j}^{\mathcal{H}(m_i)} = (\prod_{j=1, j \neq i}^{n} h_j^{\mathcal{H}(m_i)})^{z_i}$. We might omit the input $aux$ for presentation simplicity.
- $\mathsf{VerifyOpen}(vc, m_i, i, \pi_i) \rightarrow 0/1$. It checks whether $e(vc/h_i^{\mathcal{H}(m_i)}, h_i) = e(\pi_i, g)$ or not.

Regarding the *size* of the commitment and the openness proof, it is clear that they contain only a single group element in $\mathbb{G}$, which corresponds to only $\mathcal{O}(\lambda)$ bits and is independent to the length of each committed element. In addition, all algorithms run in polynomial time, and they (except the setup) are *deterministic*.

**Non-interactive threshold signature ($\mathsf{TSIG}$).** Given an established $(t, n)$-threshold signature, each party $\mathcal{P}_i$ has a private function denoted by $\mathsf{SignShare}_{(t)}(sk_i, \cdot)$ to produce its "partial" signature, and there are also three public functions $\mathsf{VerifyShare}_{(t)}$, $\mathsf{Combine}_{(t)}$ and $\mathsf{VerifyThld}_{(t)}$, which can respectively validate the "partial" signature, combine "partial" signatures into a "full" signature, and validate the "full" signature. Note the subscript $(t)$ denotes the threshold $t$ through the paper.

Formally, a non-interactive $(t, n)$-threshold signature scheme $\mathsf{TSIG}$ is a tuple of hereunder algorithms/protocols among $n$ parties $\{\mathcal{P}_i\}_{i \in [n]}$:

1. $\mathsf{TSIG.Setup}(\lambda, t, n) \rightarrow (mpk, \mathbf{pk}, \mathbf{sk})$. Given $t$, $n$ and security parameter $\lambda$, generate a special public key $mpk$, a vector of public keys $\mathbf{pk} = (pk_1, \cdots, pk_n)$, and a vector of secret keys $\mathbf{sk} = (sk_1, \cdots, sk_n)$ where $\mathcal{P}_i$ gets $sk_i$ only.
2. $\mathsf{SignShare}_{(t)}(sk_i, m) \rightarrow \sigma_i$. On input a message $m$ and a secret key share $sk_i$, this deterministic algorithm outputs a "partial" signature share $\sigma_i$. Note that the subscript $(t)$ denotes the threshold $t$.
3. $\mathsf{VerifyShare}_{(t)}(m, (i, \rho_i)) \rightarrow 0/1$. Given a message $m$, a "partial" signature $\rho_i$ and an index $i$ (along with implicit input $mpk$ and $\mathbf{pk}$), this deterministic algorithm outputs 1 (accept) or 0 (reject).
4. $\mathsf{Combine}_{(t)}(m, \{(i, \rho_i)\}_{i \in S}) \rightarrow \sigma/\perp$. Given a message $m$ and $t$ indexed partial-signatures $\{(i, \rho_i)\}_{i \in S}$ (along with implicit input $mpk$ and $\mathbf{pk}$), this algorithm outputs a "full" signature $\sigma$ for message $m$ (or $\perp$).
5. $\mathsf{VerifyThld}_{(t)}(m, \sigma) \rightarrow 0/1$. Given a message $m$ and an "aggregated" full signature $\sigma$ (along with the implicit input $mpk$), this algorithms outputs 1 (accept) or 0 (reject).

Informally, we require an established $\mathsf{TSIG}$ scheme to satisfy **correctness**, **robustness** and **unforgeability**:

- **Correctness of TSIG**. The correctness property requires that: (i) for $\forall$ $m$ and $i \in [n]$, $\Pr[\mathsf{VerifyShare}_{(t)}(m,(i,\rho_i)) = 1 \mid \rho_i \leftarrow \mathsf{SignShare}_{(t)}(sk_i, m)] = 1$; (ii) for $\forall$ $m$ and $S \subset [n]$ that $|S| = t$, $\Pr[\mathsf{VerifyThld}_{(t)}(m,\sigma) = 1 \mid \forall i \in S, \rho_i \leftarrow \mathsf{SignShare}_{(t)}(sk_i, m) \wedge \sigma \leftarrow \mathsf{Combine}_{(t)}(m, \{(i,\rho_i)\}_{i\in S})] = 1$.
- **Robustness**. No P.P.T. adversary can produce $t$ valid "partial" signature shares, s.t. running Combine over these "partial" signatures does *not* produce a valid "full" signature, except with negligible probability in $\lambda$. Intuitively, robustness ensures any $t$ valid "partial" signatures for a message must induce a valid "full" signature [39].
- **Unforgeability** (against adaptive adversary). The unforgeability can be defined by a threshold-version chosen-message game against adaptive adversary [28]. Intuitively, the unforgeability ensures that no P.P.T. adversary $\mathcal{A}$ that adaptively corrupts $f$ parties ($f < t$) can produce a valid "full" signature except with negligible probability in $\lambda$, unless $\mathcal{A}$ receives some "partial" signatures produced by $t - f$ parties that are honest.

*Instantiation.* The above adaptively secure non-interactive threshold signature can be realize due to the construction in [28], in which each "partial" signatures $\rho$ and every "full" signature $\sigma$ are $\mathcal{O}(\lambda)$ bits in length.

**Threshold common coin (Coin).** A $(t,n)$-Coin is a protocol among $n$ parties, through which any $t$ honest parties can mint a common coin $r$ uniformly sampled over $\{0,1\}^\kappa$. The adversary corrupting up to $f$ parties (where $f < t$) cannot predicate coin $r$, unless $t-f$ honest parties invoke the protocol.

Formally, an established $(t,n)$-Coin scheme satisfies the following properties except with negligible probability in $\lambda$:

- **Termination.** Once $t$ honest parties activate Coin associated to ID (denoted by Coin[ID]), each honest party that activates Coin[ID] will output a common value $r$.
- **Agreement.** If two honest parties output $r$ and $r'$ in Coin[ID] respectively, then $r = r'$.
- **Unbiasedness.** Under the influence of any P.P.T. adversary, the distribution of the outputs of Coin[$\cdot$] is computationally indistinguishable from the uniform distribution over $\{0,1\}^\kappa$.
- **Unpredictability.** A P.P.T. adversary $\mathcal{A}$ who can adaptively corrupt up to $f$ parties (e.g. $f < t$) cannot predicate the output of Coin[ID] better than guessing, unless $t-f$ honest parties activate Coin[ID].

*Instantiation.* $(t,n)$-Coin can be realized from non-interactive $(t,n)$-threshold signature due to Cachin, Kursawe and Shoup [11] in the random oracle model. Moreover, it is immediately to generalize the Coin protocol in [11] against adaptive adversary [2,29], if being given non-interactive threshold signature with adaptive security [28]. All relevant common coins in this paper are adaptively secure, and can be instantiated through the construction in [29], which incur $\mathcal{O}(\lambda n^2)$ bits, $\mathcal{O}(n^2)$ messages and constant $\mathcal{O}(1)$ running time, where $\lambda$ is the cryptographic security parameter.

**Identity election**. In our context, an identity Election protocol is a $(2f+1,n)$-Coin protocol that returns a common value over $\{1,\cdots,n\}$. Through the paper, this particular Coin is under the descriptive alias Election, which is also a standard term due to Ben-Or and El-Yaniv [4].

**Asynchronous binary agreement (ABA).** In an asynchronous binary agreement (ABA) protocol among $n$ parties, the honest parties input a single bit, and aim to output a common bit $b \in \{0,1\}$ which shall be input of at least one honest party.

Formally, an ABA protocol satisfies the properties except with negligible probability:

- **Termination**. If all honest parties receive binary inputs in $\{0,1\}$ to activate ABA associated to ID (denoted by ABA[ID]), each honest party outputs a bit for ID.
- **Agreement**. If any two honest parties output $b$ and $b'$ for ID receptively, then $b = b'$.
- **Validity**. If any honest party outputs a bit $b \in \{0,1\}$ for ID, then at least one honest party inputs $b$ for ID.

*Instantiation.* Given adaptively secure non-interactive threshold signature in [28], the Coin scheme in [11] can immediately be generalized to defend against adaptive adversary [29], which further generalizes many ABA constructions [32,10] to be adaptively secure [29]. In particular, we adopt

the $\mathsf{ABA}$ secure against adaptive adversary controlling up to $\lfloor \frac{n-1}{3} \rfloor$ parties in [29], which attains expected $\mathcal{O}(1)$ running time, asymptomatic $\mathcal{O}(n^2)$ messages and $\mathcal{O}(\lambda n^2)$ bits, where $\lambda$ is the cryptographic security parameter.

### 4.2 Other notations

We use $\langle x, y \rangle$ to denote a string concatenating two strings $x$ and $y$. Any message between two parties is of the form $(\textsc{MsgType}, \mathsf{ID}, \dots)$, where $\mathsf{ID}$ represents the identifier that tags the protocol instance and $\textsc{MsgType}$ specifies the message type. Moreover, $\Pi[\mathsf{ID}]$ refers to an instance of the protocol $\Pi$ with identifier $\mathsf{ID}$, and $y \leftarrow \Pi[\mathsf{ID}](x)$ means to invoke $\Pi[\mathsf{ID}]$ on input $x$ and obtain $y$ as output.

**Probabilistically uniformly bounded statistic**. Here we showcase the definition of uniformly-bounded statistic, which is a conventional notion [10,2] to rigorously describe the performance metrics of fault-tolerant protocols under the influence of arbitrary adversary, such as the messages and communicated bits among honest parties.

**Definition 2.** ***Uniformly Bounded Statistic***. *Let $X$ be a random variable representing a protocol statistic (for example, the number of messages generated by the honest parties during the protocol execution). We say that $X$ is probabilistically uniformly bounded, if there exists a fixed polynomial $T(\lambda)$ and a fixed negligible function $\delta(k)$, such that for any adversary $\mathcal{A}$ (of the protocol), there exists a negligible function $\epsilon(\lambda)$ for all $\lambda \geq 0$ and $k \geq 0$,*

$$\Pr[X \geq kT(\lambda)] \leq \delta(k) + \epsilon(\lambda).$$

A probabilistically uniformly bounded statistic of a protocol performance metric $X$ cannot exceed the uniform bound except with negligible probability, independent of the adversary. More precisely, this means, there exists a constant $c$, s.t. for any adversary $\mathcal{A}$, the expected value of $X$ must be bounded by $cT(k) + \epsilon'(\lambda)$, where $\epsilon'(\lambda)$ is a negligible function.

## 5 Asynchronous Provable Dispersal Broadcast

The dominating $O(\ell n^2)$ term in the communication complexity of existing $\mathsf{MVBA}$ protocols [10,2] is because every party broadcasts its own input *all* other parties. This turns out to be unnecessary, as in the $\mathsf{MVBA}$ protocol, only one single party's input is decided as output. To remedy the needless communication overhead in $\mathsf{MVBA}$, we introduce a new *dispersal-then-recast* methodology, through which each party $\mathcal{P}_i$ only has to spread the coded *fragments* of its input $v_i$ to every other party instead of its entire input.

This section introduces the core building block, namely, the asynchronous provable dispersal broadcast ($\mathsf{APDB}$), to instantiate the *dispersal-then-recast* idea. The notion is carefully tailored to be efficiently implementable. Especially, in contrast to related $\mathsf{AVID}$ protocols [12,23], $\mathsf{APDB}$ can disperse a value at a cost of linear messages instead of $\mathcal{O}(n^2)$, as a reflection of following trade-offs:

- The $\mathsf{APDB}$ notion weakens $\mathsf{AVID}$, so upon that a party outputs a coded fragment in the dispersal instance of $\mathsf{APDB}$, there is no guarantee that other parties will output the consistent fragments. Thus, it could be not enough to recover the dispersed value by only $f+1$ honest parties, as these parties might receive (probably inconsistent) fragments.
- To compensate the above weakenings, we let the sender to spread the coded fragments of its input along with a succinct vector commitment of all these fragments, and then produce two succinct "proofs" *lock* and *done*. The "proofs" facilitate: (i) the *lock* proof ensures that $2f+1$ parties receive some fragments that are committed in the same vector commitment, so the honest parties can either recover the same value, or output $\perp$ (that means the committed fragments are inconsistent); (ii) the *done* proof ensures that $2f+1$ parties deliver valid *lock*s, thus allowing the parties to reach a common decision, e.g., via a (biased) binary $\mathsf{BA}$ [10], to all agree to jointly recover the dispersed value, which makes the value deemed to be recoverable.

In this way, the overall communication of dispersing a value can be brought down to minimum as the size of each fragment is only $\mathcal{O}(\ell/n)$ where $\ell$ is the length of input $v$. Moreover, this well-tuned notion can be easily implemented in light of [2] and costs only linear messages. These efficiencies are needed to achieve the optimal communication and message complexities for MVBA.

**Defining asynchronous provable dispersal broadcast.** Formally, the syntax and properties of a APDB protocol are defined as follows.

**Definition 3.** *An* APDB *protocol with a sender $\mathcal{P}_s$ consists of a provable dispersal subprotocol (*PD*) and a recast subprotocol (*RC*) with a pair of validation functions* (ValidateLock, ValidateDone)*:*

- **PD subprotocol**. *In the* PD *subprotocol (with identifier* ID*) among $n$ parties, a designated sender $\mathcal{P}_s$ inputs a value $v \in \{0,1\}^\ell$, and aims to split $v$ into $n$ encoded fragments and disperses each fragment to the corresponding party. During the* PD *subprotocol with identifier* ID*, each party is allowed to invoke an abandon*(ID) *function. After* PD *terminates, each party shall output two strings store and lock, and the sender shall output an additional string done.*
  *Note that the lock and done strings are said to be valid for the identifier* ID*, if and only if* ValidateLock(ID, $lock$) $= 1$ *and* ValidateDone(ID, $done$) $= 1$*, respectively.*
- **RC subprotocol**. *In the* RC *subprotocol (with identifier* ID*), all honest parties take the output of the* PD *subprotocol (with the same* ID*) as input, and aim to output the value $v$ that was dispersed in the* RC *subprotocol. Once* RC *is completed, the parties output a common value in* $\{0,1\}^\ell \cup \bot$.

*An* APDB *protocol (*PD, RC*) with identifier* ID *satisfies the following properties except with negligible probability:*

- **Termination**. *If the sender $\mathcal{P}_s$ is honest and all honest parties activate* PD[ID] *without invoking abandon*(ID)*, then each honest party would output store and valid lock for* ID*; additionally, the sender $\mathcal{P}_s$ outputs valid done for* ID.
- **Recast-ability**. *If all honest parties invoke* RC[ID] *with inputting the output of* PD[ID] *and at least one honest party inputs a valid lock, then: (i) all honest parties recover a common value; (ii) if the sender dispersed $v$ in* PD[ID] *and has not been corrupted before at least one party delivers valid lock, then all honest parties recover $v$ in* RC[ID].
  *Intuitively, the* recast-ability *captures that the valid lock is a "proof" attesting that the input value dispersed via* PD[ID] *can be consistently recovered by* all *parties through collectively running the corresponding* RC[ID] *instance.*
- **Provability**. *If the sender of* PD[ID] *produces valid done, then at least $f+1$ honest parties output valid lock.*
  *Intuitively, the* provability *indicates that done is a "completeness proof" attesting that at least $f+1$ honest parties output valid locks, such that the parties can exchange locks and then vote via* ABA *to reach an agreement that the dispersed value is deemed recoverable.*
- **Abandon-ability**. *If every party (and the adversary) cannot produce valid lock for* ID *and $f+1$ honest parties invoke abandon*(ID)*, no party would deliver valid lock for* ID.

### 5.1 Overview of the APDB protocol

For the PD subprotocol with identifier ID, it has a simple structure of four one-to-all or all-to-one rounds: sender $\xrightarrow{\text{Store}}$ parties $\xrightarrow{\text{Stored}}$ sender $\xrightarrow{\text{Lock}}$ parties $\xrightarrow{\text{Locked}}$ sender. Through a Store message, every party $\mathcal{P}_i$ receives $store := \langle vc, m_i, i, \pi_i \rangle$, where $m_i$ is an encoded fragment of the sender's input, $vc$ is a (deterministic) commitment of the vector of all fragments, and $\pi_i$ attests $m_i$'s inclusion in $vc$ at the $i$-th position; then, through Stored messages, the parties would give the sender "partial" signatures for the string $\langle$Stored, ID, $vc\rangle$; next, the sender combines $2f+1$ valid "partial" signatures, and sends every party the combined "full" signature $\sigma_1$ for the string $\langle$Stored, ID, $vc\rangle$ via Locked messages, so each party can deliver $lock := \langle vc, \sigma_1 \rangle$; finally, each party sends a "partial" signature for the string $\langle$Locked, ID, $vc\rangle$, such that the sender can again combine the "partial" signatures to produce a valid "full" signature $\sigma_2$ for the string $\langle$Locked, ID, $vc\rangle$, which allows the sender to deliver $done := \langle vc, \sigma_2 \rangle$.

For the RC subprotocol, it has only one-round structure, as each party only has to take some output of PD subprotocol as input (i.e., *lock* and *store*), and multicasts these inputs to all parties. As long as an honest party inputs a valid *lock*, there are at least $f+1$ honest parties deliver valid *store*s that are bound to the vector commitment $vc$ included in *lock*, so all parties can eventually reconstruct the dispersed value that was committed in the commitment $vc$.

**Algorithm 1** Validation func of APDB protocol, with identifier ID

**function** ValidateStore($i'$, *store*): ▷ ValidateStore verifies *store* commits a fragment $m_i$ in $vc$ at $i$-th position
1: parse *store* as $\langle vc, i, m_i, \pi_i \rangle$
2: return VerifyOpen$(vc, m_i, i, \pi_i) \wedge i = i'$

**function** ValidateLock(ID, *lock*): ▷ ValidateLock validates *lock* contains a commitment $vc$ signed by $2f+1$ parties
3: parse *lock* as $\langle vc, \sigma_1 \rangle$
4: return VerifyThld$_{(2f+1)}(\langle \text{STORED}, \text{ID}, vc \rangle, \sigma_1)$

**function** ValidateDone(ID, *done*): ▷ ValidateDone validates *done* to attest $2f+1$ parties receive valid *lock*
5: parse *done* as $\langle vc, \sigma_2 \rangle$
6: return VerifyThld$_{(2f+1)}(\langle \text{LOCKED}, \text{ID}, vc \rangle, \sigma_2)$

### 5.2 Details of the APDB protocol

As illustrated in Alg 1, the APDB protocol is designed with a few functions called as ValidateStore, ValidateLock and ValidateDone to validate *done*, *lock* and *store*, respectively. ValidateStore is to check the *store* received by the party $\mathcal{P}_i$ includes a fragment $m_i$ that is committed in a vector commitment $vc$ at the $i$-th position, ValidateLock validates *lock* to verify that $2f+1$ parties (i.e., at least $f+1$ honest parties) receive the fragments that are correctly committed in the same vector commitment $vc$, and ValidateDone validates *done* to verify that $2f+1$ parties (i.e., at least $f+1$ honest parties) have delivered valid *lock*s (that contain the same $vc$).

PD *subprotocol*. The details of the PD subprotocol are shown in Algorithm 2. In brief, a PD instance with identifier ID (i.e., PD[ID]) allows a designated sender $\mathcal{P}_s$ to disperse a value $v$ as follows:

1. *Store-then-Stored* (line 1-6, 13-15, 19-23). When the sender $\mathcal{P}_s$ receives an input value $v$ to disperse, it encodes $v$ to generate a vector of coded fragments $m = (m_1, \ldots, m_n)$ by an $(f+1, n)$-erasure code; then, $\mathcal{P}_s$ commits $m$ in a vector commitment $vc$. Then $\mathcal{P}_s$ sends *store* including the commitment $vc$, the $i$-th coded fragment $m_i$ and the commitment opening $\pi_i$ to each party $\mathcal{P}_i$ by STORE messages. Upon receiving (STORE, ID, *store*) from the sender, $\mathcal{P}_i$ verifies whether *store* is valid. If that is the case, $\mathcal{P}_i$ delivers *store* and sends a $(2f+1, n)$-partial signature $\rho_{1,i}$ for $\langle \text{STORED}, \text{ID}, vc \rangle$ back to the sender through a STORED message.
2. *Lock-then-Locked* (line 7-9, 16-18, 24-28). Upon receiving $2f+1$ valid STORED messages from distinct parties, the sender $\mathcal{P}_s$ produces a full signature $\sigma_1$ for the string $\langle \text{STORED}, \text{ID}, vc \rangle$. Then, $\mathcal{P}_s$ sends *lock* including $vc$ and $\sigma_1$ to all parties through LOCK messages. Upon receiving LOCK message, $\mathcal{P}_i$ verifies whether $\sigma_1$ is deemed as a valid full signature. If that is the case, $\mathcal{P}_i$ delivers $lock = \langle vc, \sigma_1 \rangle$, and sends a $(2f+1, n)$-partial signature $\rho_{2,i}$ for the string $\langle \text{LOCKED}, \text{ID}, vc \rangle$ back to the sender through a LOCKED message.
3. *Done* (line 10-12). Once the sender $\mathcal{P}_s$ receives $2f+1$ valid LOCKED messages from distinct parties, it produces a full signature $\sigma_2$ for $\langle \text{LOCKED}, \text{ID}, vc \rangle$. Then $\mathcal{P}_s$ outputs the completeness proof $done = \langle vc, \sigma_2 \rangle$ and terminates the dispersal.
4. *Abandon* (line 29). A party can invoke *abandon*(ID) to explicitly stop its participation in this dispersal instance with identification ID. In particular, if $f+1$ honest parties invoke *abandon*(ID), the adversary can no longer corrupt the sender of PD[ID] to disperse anything across the network.

**Algorithm 2** PD subprotocol, with identifier ID and sender $\mathcal{P}_s$

**let** $S_1 \leftarrow \{\ \}$, $S_2 \leftarrow \{\ \}$, $stop \leftarrow 0$

/* Protocol for the sender $\mathcal{P}_s$ */

1: **upon** receiving an input value $v$ **do**
2: $m \leftarrow \mathsf{Enc}(v)$, where $v$ is parsed as a $f+1$ vector and $m$ is a $n$ vector
3: $vc \leftarrow \mathsf{VCom}(m)$
4: **for** each $j \in [n]$ **do**
5: $\pi_j \leftarrow \mathsf{Open}(vc, m_j, j)$
6: **let** $store := \langle vc, m_i, i, \pi_i \rangle$ and **send** $(\textsc{Store}, \mathsf{ID}, store)$ to $\mathcal{P}_j$ ▷ multicast *store*
7: **wait** until $|S_1| = 2f+1$
8: $\sigma_1 \leftarrow \mathsf{Combine}_{(2f+1)}(\langle \textsc{Stored}, \mathsf{ID}, vc \rangle, S_1)$
9: **let** $lock := \langle vc, \sigma_1 \rangle$ and **multicast** $(\textsc{Lock}, \mathsf{ID}, lock)$ to all parties ▷ multicast *lock* proof
10: **wait** until $|S_2| = 2f+1$
11: $\sigma_2 \leftarrow \mathsf{Combine}_{(2f+1)}(\langle \textsc{Locked}, \mathsf{ID}, vc \rangle, S_2)$
12: **let** $done := \langle vc, \sigma_2 \rangle$ and **deliver** *done* ▷ produce *done* proof
13: **upon** receiving $(\textsc{Stored}, \mathsf{ID}, \rho_{1,j})$ from $\mathcal{P}_j$ for the first time **do**
14: **if** $\mathsf{VerifyShare}_{(2f+1)}(\langle \textsc{Stored}, \mathsf{ID}, vc \rangle, (j, \rho_{1,j})) = 1$ and $stop = 0$ **then**
15: $S_1 \leftarrow S_1 \cup (j, \rho_{1,j})$
16: **upon** receiving $(\textsc{Locked}, \mathsf{ID}, \rho_{2,j})$ from $\mathcal{P}_j$ for the first time **do**
17: **if** $\mathsf{VerifyShare}_{(2f+1)}(\langle \textsc{Locked}, \mathsf{ID}, vc \rangle, (j, \rho_{2,j})) = 1$ and $stop = 0$ **then**
18: $S_2 \leftarrow S_2 \cup (j, \rho_{2,j})$

/* Protocol for each party $\mathcal{P}_i$ */

19: **upon** receiving $(\textsc{Store}, \mathsf{ID}, store)$ from sender $\mathcal{P}_s$ for the first time **do**
20: **if** $\mathsf{ValidateStore}(i, store) = 1$ and $stop = 0$ **then**
21: **deliver** *store* and parse it as $\langle vc, i, m_i, \pi_i \rangle$ ▷ receive *store*
22: $\rho_{1,i} \leftarrow \mathsf{SignShare}_{(2f+1)}(sk_i, \langle \textsc{Stored}, \mathsf{ID}, vc \rangle)$
23: **send** $(\textsc{Stored}, \mathsf{ID}, \rho_{1,i})$ to $\mathcal{P}_s$
24: **upon** receiving $(\textsc{Lock}, \mathsf{ID}, lock)$ from sender $\mathcal{P}_s$ for the first time **do**
25: **if** $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$ and $stop = 0$ **then**
26: **deliver** *lock* and parse it as $\langle vc, \sigma_1 \rangle$ ▷ receive *lock*
27: $\rho_{2,i} \leftarrow \mathsf{SignShare}_{(2f+1)}(sk_i, \langle \textsc{Locked}, \mathsf{ID}, vc \rangle)$
28: **send** $(\textsc{Locked}, \mathsf{ID}, \rho_{2,i})$ to $\mathcal{P}_s$

---

**procedure** *abandon*(ID):
29: $stop \leftarrow 1$

RC *subprotocol*. The construction of the RC subprotocol is shown in Algorithm 3. The input of RC subprotocol consists of *lock* and *store*, which were probably delivered during the PD subprotocol. In brief, the execution of a RC instance with identification ID is as:

1. *Recast* (line 1-5). If the party $\mathcal{P}_i$ inputs *lock* and/or *store*, it multicasts them to all parties.
2. *Deliver* (line 6-18). If the party $\mathcal{P}_i$ receives a valid *lock* message, it waits for $f+1$ valid *store*s bound to this *lock*, such that $\mathcal{P}_i$ can reconstruct a value $v$ (or a special symbol $\perp$).

**Algorithm 3** RC subprotocol with identifier ID, for each party $\mathcal{P}_i$

```
    let C ← [ ]      ▷ a dictionary structure C[vc] that stores the set of coded fragments committed in vc
 1: upon receiving input (store, lock) do                                  ▷ multicast lock and/or store
 2:     if lock ≠ ∅ then
 3:         multicast (RcLock, ID, lock) to all
 4:     if store ≠ ∅ then
 5:         multicast (RcStore, ID, store) to all
 6: upon receiving (RcLock, ID, lock) do
 7:     if ValidateLock(ID, lock) = 1 then    ▷ assert: only one valid lock for each ID (c.f. Lemma 3)
 8:         multicast (RcLock, ID, lock) to all, if was not sent before
 9:         parse lock as ⟨vc, σ1⟩
10:         wait until |C[vc]| = f + 1
11:             v ← Dec(C[vc])                        ▷ interpolate to decode the erasure code
12:             if VCom(Enc(v)) = vc then return v    ▷ deliver the dispersed value (c.f. Corollary 1)
13:             else return ⊥
14: upon receiving (RcStore, ID, store) from P_j for the first time do
15:     if ValidateStore(j, store) = 1 then
16:         parse store as ⟨vc, m_j, j, π_j⟩
17:         C[vc] ← C[vc] ∪ (j, m_j)                  ▷ record fragments committed to each vc
18:     else discard the invalid message
```

### 5.3 Analyses of Asynchronous Provable Dispersal Broadcast

Here we present the detailed proofs along with the complexity analyses for APDB protocol.

**Security intuition**. The tuple of protocols in Algorithm 2 and 3 realize APDB among $n$ parties against the adaptive adversary controlling up to $f \leq \lfloor \frac{n-1}{3} \rfloor$ parties, given (i) $(f+1, n)$-erasure code, (ii) deterministic $n$-vector commitment with the position-binding property, and (iii) established $(2f+1, n)$-threshold signature with adaptive security. The high-level intuition is: (i) if any honest party outputs valid *lock*, then at least $f+1$ honest parties receives the code fragments committed in the same vector commitment, and the position-binding property ensures that the honest parties can collectively recover a common value (or the common $\perp$) from these committed fragments; (ii) whenever any party can produce a valid *done*, it attests that $2f+1$ (namely, at least $f+1$ honest) parties have indeed received valid *lock*s.

**The proofs of APDB**. Now we prove the Algorithm 2 and 3 would satisfy the properties of APDB as defined in Definition 3 with all but negligible probability.

**Corollary 1.** *Considering a $n$-vector commitment scheme* (VCom, Open, VerifyOpen) *and a $(k, n)$-erasure coding scheme* (Enc, Dec), *we have the following observations:*

- *Correctness. For any $S \subset [n]$ that $|S| = k$,* $\Pr[\mathsf{Dec}(\{(i, m_i)\}_{j \in S}) = v \land \mathsf{VCom}(\mathsf{Enc}(v)) = vc \mid (m_1, \cdots, m_n) \leftarrow \mathsf{Enc}(v) \land vc \leftarrow \mathsf{VCom}((m_1, \cdots, m_n))] = 1$. *The property reveals: if*

*a vector of the coded fragments of a value $v$ are committed to a vector commitment $vc$, then any $k$-subset of the fragments can recover the original value $v$, whose encoded fragments can be used to produce the same commitment $vc$. This is true, because of the correctness of erasure coding and the determinism of* $\mathsf{VCom}$.

- *Security. For any $S_1 \subset [n]$ and $S_2 \subset [n]$ that $|S_1| = |S_2| = k$ and $S_1 \neq S_2$, it is computationally infeasible for any P.P.T. adversary (on input the security parameter $\lambda$ and all other public system parameters) to produce $vc$, $\{(i, m_i, \pi_i)\}_{i \in S_1}$ and $\{(j, m_j, \pi_j)\}_{j \in S_2}$ where $\forall i \in S_1, \mathsf{VerifyOpen}(vc, m_i, i, \pi_i) = 1$ and $\forall j \in S_2, \mathsf{VerifyOpen}(vc, m_j, j, \pi_j) = 1$, such that: (i) $\mathsf{VCom}(\mathsf{Enc}(\mathsf{Dec}(\{(i, m_i)\}_{i \in S_1}))) = vc \quad \wedge \quad \mathsf{VCom}(\mathsf{Enc}(\mathsf{Dec}(\{(j, m_j)\}_{j \in S_2}))) = vc \quad \wedge$ $\mathsf{Dec}(\{(i, m_i)\}_{i \in S_1}) \neq \mathsf{Dec}(\{(j, m_j)\}_{j \in S_2})$, or (ii) $\mathsf{VCom}(\mathsf{Enc}(\mathsf{Dec}(\{(i, m_i)\}_{i \in S_1}))) = vc \quad \wedge$ $\mathsf{VCom}(\mathsf{Enc}(\mathsf{Dec}(\{(j, m_j)\}_{j \in S_2}))) \neq vc$. The property indicates whenever the (probably malicious generated) coded fragments are committed to $vc$, any two $k$-subsets of these committed fragments must: either consistently recover a common value that can re-produce the same commitment $vc$, or recover some values that simultaneously re-produce some commitments different from $vc$.*

**Lemma 1. Termination**. *If the sender $\mathcal{P}_s$ is honest and all honest parties activate* $\mathsf{PD}[\mathsf{ID}]$ *without invoking abandon*($\mathsf{ID}$)*, then each honest party would output store and valid lock for* $\mathsf{ID}$ *s.t.* $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$*; additionally, the sender $\mathcal{P}_s$ outputs valid done for* $\mathsf{ID}$.

*Proof.* In case all honest parties activate $\mathsf{PD}[\mathsf{ID}]$ without abandoning, all honest parties will follow the $\mathsf{PD}$ protocol specified by the pseudocode shown in Algorithm 2. In addition, since the sender $\mathcal{P}_s$ is honest, it also follows the protocol.

Due to the $\mathsf{PD}$ protocol, the honest sender firstly sends $(\textsc{Store}, \mathsf{ID}, store)$ to $\mathcal{P}_i$, where the Store message satisfies $\mathsf{ValidateStore}(i, store) = 1$; after receiving the valid Store message, all honest parties will send Stored messages back to the sender. When $\mathcal{P}_s$ receiving $2f+1$ (the number of honest parties at least is $2f+1$) valid Stored messages, $\mathcal{P}_s$ will combine these messages to generate valid signature $\sigma_1$ by $\mathsf{Combine}_{(2f+1)}$ algorithm and then obtain a valid *lock*. After that, $\mathcal{P}_s$ will send $(\textsc{Lock}, \mathsf{ID}, lock)$ to all, where the Lock message satisfies $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$.

Next, after receiving the valid Lock message, all honest parties will send Locked message back to the sender. When receiving $2f+1$ valid Locked message, the sender can generate valid signature $\sigma_2$ by $\mathsf{Combine}_{(2f+1)}$ algorithm. Hence, the honest sender $\mathcal{P}_s$ can outputs a completeness proof $done = \langle vc, \sigma_2 \rangle$, s.t. $\mathsf{ValidateDone}(\mathsf{ID}, done) = 1$.

**Lemma 2. Provability**. *If the sender of* $\mathsf{PD}[\mathsf{ID}]$ *can produce done s.t.* $\mathsf{ValidateDone}(\mathsf{ID}, done) = 1$*, then at least $f+1$ honest parties output lock s.t.* $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$.

*Proof.* $\mathsf{ValidateDone}(\mathsf{ID}, done) = 1$ is equivalent to that $\mathsf{VerifyThld}_{(2f+1)}(\langle \textsc{Locked}, \mathsf{ID}, vc \rangle, \sigma_2) = 1$ due to Algorithm 1, which means that without overwhelming probability, the sender does receive at least $2f+1$ Locked message from distinct $\mathcal{P}_j$ to generate $\sigma_2$ (otherwise the threshold signature can be forged). With all but negligible probability, at least $f+1$ honest parties send Locked messages to the sender with attaching their "partial" signatures for $\langle \textsc{Locked}, \mathsf{ID}, vc \rangle$. From Algorithm 2, we know the honest parties sends their "partial" signatures for $\langle \textsc{Locked}, \mathsf{ID}, vc \rangle$ to the sender, iff they deliver valid *lock* which satisfies $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$. So this lemma holds with overwhelming probability, otherwise it would break the unforgeability of the underlying threshold signature.

**Lemma 3.** *If two parties $\mathcal{P}_i$ and $\mathcal{P}_j$ deliver lock and lock$'$ in* $\mathsf{RC}[\mathsf{ID}]$ *and* $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$ $\wedge$ $\mathsf{ValidateLock}(\mathsf{ID}, lock') = 1$*, then lock = lock$'$.*

*Proof.* When $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$, we can assert that $\mathsf{VerifyThld}_{(2f+1)}(\langle \textsc{Stored}, \mathsf{ID}, vc \rangle, \sigma_1) = 1$ due to Algorithm 1. According to the Algorithm 2, $\sigma_1$ was generated by combining $2f+1$ distinct parties' partial signatures for $\langle \textsc{Stored}, \mathsf{ID}, vc \rangle$. Therefore, there have at least $f+1$ honest parties produced a share signature for $(\textsc{Stored}, \mathsf{ID}, vc)$.

However, if $\mathsf{ValidateLock}(\mathsf{ID}, lock') = 1$, $\mathsf{VerifyThld}_{(2f+1)}(\langle \textsc{Stored}, \mathsf{ID}, vc' \rangle, \sigma_1') = 1$ also holds, which means that there are at least $f+1$ honest parties that also produce a share signature for $(\textsc{Stored}, \mathsf{ID}, vc')$. Hence, at least one honest party produce two different Stored message if $lock \neq lock'$. However, every honest parties compute the share signature for Stored message at most once for a given identifier $\mathsf{ID}$. Hence, we have $lock = lock'$ with overwhelming probability; otherwise, the unforgeability of the underlying threshold signature would be broken.

**Lemma 4. Recast-ability**. *If all honest parties invoke* $\mathsf{RC}[\mathsf{ID}]$ *with inputting the output of* $\mathsf{PD}[\mathsf{ID}]$ *and at least one honest party inputs a valid lock, then: (i) all honest parties recover a common value; (ii) if the sender dispersed* $v$ *in* $\mathsf{PD}[\mathsf{ID}]$ *and has not been corrupted before at least one party delivers valid lock, then all honest parties recover* $v$ *in* $\mathsf{RC}[\mathsf{ID}]$.

*Proof.* To prove the conclusion (i) of the Lemma, we would prove the following two statements: first, all honest parties can output a value; second, the output of any two honest parties would be same.

**Part 1:** Since at least one honest party delivers *lock* satisfying $\mathsf{ValidateLock}(\mathsf{ID}, lock) = 1$, according to the Algorithm 3, it will multicast $(\textsc{RcLock}, \mathsf{ID}, lock)$ to all. Hence, all honest parties can receive the valid *lock*.

Note whenever a valid $lock := \langle vc, \sigma_1 \rangle$ can be produced, there is a valid threshold signature $\sigma_1$ was generated by combining $2f + 1$ distinct parties' partial signatures for $(\textsc{Store}, \mathsf{ID}, vc)$, due to Algorithm 2. Also notice that an honest party partially signs $(\textsc{Store}, \mathsf{ID}, vc)$, iff it delivers valid *store* that is bound to the commitment $vc$. So there are at least $f + 1$ honest parties deliver the valid *store* that are committed to the same commitment string $vc$.

Thus there have at least $f + 1$ honest parties $\mathcal{P}_i$ will multicast valid $(\textsc{RcStore}, \mathsf{ID}, store)$ message to all, due to Algorithm 3. For each honest party, they can eventually receive a valid valid $\textsc{RcLock}$ message and $f + 1$ valid $\textsc{RcStore}$ messages, that are corresponding to the same $vc$. So all parties will always attempt the decode the received fragments in the $\textsc{RcStore}$ messages to eventually recover some value.

**Part 2:** From the Lemma 3, all honest parties would receive valid $\textsc{RcLock}$ messages with the same $lock := \langle vc, \sigma_1 \rangle$. Therefore, each honest party can receive $f + 1$ valid $\textsc{RcStore}$ messages, which contain $f + 1$ fragments that are committed to the same vector commitment $vc$. Due to Corollary 1, either every honest party $\mathcal{P}_i$ have $\mathsf{VCom}(\mathsf{Enc}(\mathsf{Dec}(C[vc]))) = vc$ or every honest party $\mathcal{P}_i$ have $\mathsf{VCom}(\mathsf{Enc}(\mathsf{Dec}(C[vc]))) \neq vc$. Therefore, either all honest parties return a common value $\mathsf{Dec}(C[vc])$ in $\{0, 1\}^\ell$, or they return a special symbol $\perp$.

The conclusion (i) of the Lemma holds immediately by following Part 1 and Part 2.

For the conclusion (ii) of the Lemma, if the sender $\mathcal{P}_s$ has not been corrupted (so-far-uncorrupted) before at least one party delivers valid *lock* and passed the value $v$ into $\mathsf{PD}[\mathsf{ID}]$ as input, the sender would at least follow the protocol to send $\textsc{Store}$ messages for dispersing $v$. Moreover, when the so-far-uncorrupted $\mathcal{P}_s$ delivers valid *lock*, at least $f + 1$ honest parties already receive the $\textsc{Store}$ messages for dispersing $v$, so the adversary can no longer corrupts $\mathcal{P}_s$ to disperse a value $v'$ different from $v$, as it cannot produce valid *lock* or valid *done* for $v'$. From the proving of conclusion (i), we know all parties would recover the value $\mathsf{Dec}(C[vc])$, which must be $v$ due to the properties of erasure code and vector commitment.

**Lemma 5. Abandon-ability**. *If every party (and the adversary) cannot produce valid lock for* $\mathsf{ID}$ *and* $f + 1$ *honest parties invoke abandon*$(\mathsf{ID})$*, no party would deliver valid lock for* $\mathsf{ID}$.

*Proof.* From Algorithm 2, we know it needs $2f + 1$ valid $\textsc{Stored}$ messages to produce a valid $lock := \langle vc, \sigma_1 \rangle$. Since any parties (including the adversary) has not yet produced a valid *lock* and $f+1$ honest parties invoke *abandon*$(\mathsf{ID})$, there are at most $2f$ parties are participating in the $\mathsf{PD}[\mathsf{ID}]$ instance. So there are at most $2f$ valid $\textsc{Stored}$ messages, which are computationally infeasible for any party to produce a valid *lock*; otherwise, the unforgeability of underlying threshold signature would not hold.

**Theorem 2.** *The tuple of protocols described by Algorithms 2 and 3 solves asynchronous provable dispersal broadcast (*$\mathsf{APDB}$*) among* $n$ *parties against an adaptive adversary controlling* $f < n/3$ *parties, given (i)* $(f + 1, n)$*-erasure code, (ii)* $n$*-vector commitment scheme, and (iii) established non-interactive* $(2f + 1, n)$*-threshold signature with adaptive security.*

*Proof.* Lemma 1, 2, 4 and 5 complete the proof.

**The complexity analysis of APDB**. Through this paper, we consider $\ell$ is the input length and $\lambda$ is cryptographic security parameter (the length of signature, vector commitment, and openness proof for commitment), then:

- **PD complexities:** According to the process of Algorithm 2, the PD subprotocol has 4 one-to-all (or all-to-one) rounds. Hence, the total number of messages sent by honest parties is at most $4n$, which attains $\mathcal{O}(n)$ messages complexity and $\mathcal{O}(1)$ running time. Besides, the maximal size of messages is $\mathcal{O}(\ell/n + \lambda)$, so the communication complexity of PD is $\mathcal{O}(\ell + n\lambda)$.
- **RC complexities:** According to the process of Algorithm 3, the message exchanges appear in two places. First, all parties multicast the RcLock messages to all, so the first parts' messages complexity is $\mathcal{O}(n^2)$; second, all parties multicast the RcStore messages to all, thus the second parts incurring $\mathcal{O}(n^2)$ messages. Hence, the RC incurs $\mathcal{O}(n^2)$ messages complexity and constant running time. Besides, each RcLock message is sized to $\mathcal{O}(\lambda)$-bit, and the size of each RcStore message is $\mathcal{O}(\ell/n+\lambda)$-bit, so the communication complexity of RC is $\mathcal{O}(n^2\lambda)+\mathcal{O}(n\ell + n^2\lambda)$=$\mathcal{O}(n\ell + n^2\lambda)$.

## 6 An Optimal MVBA protocol

We now apply our *dispersal-then-recast* methodology to design the optimal MVBA protocol Dumbo-MVBA, using APDB and ABA. It is secure against adaptively corrupted $\lfloor\frac{n-1}{3}\rfloor$ parties, and attains optimal running time and message complexity; in addition, it costs $\mathcal{O}(\ell n + \lambda n^2)$ bits, which is asymptotically better than all previous results [2,10] and optimal for sufficiently large input.

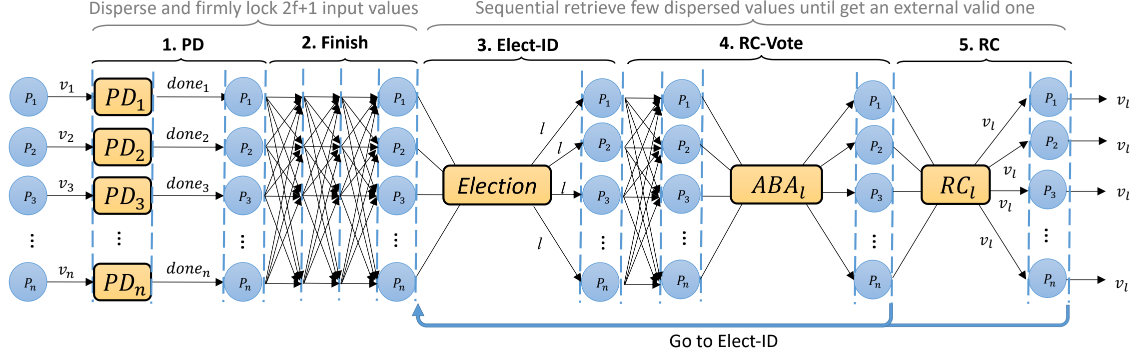


**Fig. 1.** The execution flow of Dumbo-MVBA.

### 6.1 Overview of the Dumbo-MVBA protocol

As illustrated in Figure 1, the basic ideas of our Dumbo-MVBA protocol are: (i) the parties disperse their own input values through $n$ concurrent PD instances, until they consistently realize that enough *done*s proofs for the PD instances (i.e., $2n/3$) have been produced, so they can make sure that enough honest input values (i.e., $n/3$) have been firmly locked across the network; (ii) eventually, the parties can exchange *done*s proofs to explicitly stop all PD instances; (iii) then, the parties can invoke a common coin protocol Election to randomly elect a PD instance; (iv) later, the parties exchange their *lock* proofs of the elected PD instance and then leverage ABA to vote on whether to invoke the corresponding RC instance to recast the elected dispersal; (v) when ABA returns 1, all parties would activate the RC instance and might probably recast a common value that is externally valid; otherwise (i.e., either ABA returns 0 or RC recasts invalid value), they repeat Election, until an externally valid value is elected and collectively reconstructed.

### 6.2 Details of the Dumbo-MVBA protocol

Our Dumbo-MVBA protocol invokes the following modules: (i) asynchronous provable dispersal broadcast APDB := (PD, RC); (ii) asynchronous binary agreement ABA against adaptive adversary;

(iii) $(f+1, n)$ threshold signature with adaptive security; and (iv) adaptively secure $(2f+1, n)$-Coin scheme (in the alias Election) that returns random numbers over $[n]$.

Each instance of the underlying modules can be tagged by a unique extended identifier ID. These explicit IDs extend id and are used to distinguish multiple activated instances of every underlying module. For instance, $(\mathsf{PD}[\mathsf{ID}], \mathsf{RC}[\mathsf{ID}])$ represents a pair of (PD, RC) instance with identifier ID, where $\mathsf{ID} := \langle \mathsf{id}, i \rangle$ extends the identification id to represent a specific APDB instance with a designated sender $\mathcal{P}_i$. Similarly, ABA[ID] represents an ABA instance with identifier ID, where $\mathsf{ID} := \langle \mathsf{id}, k \rangle$ and $k \in \{1, 2, \dots\}$.

**Protocol execution**. Hereunder we are ready to present the detailed protocol description (as illustrated in Algorithm 4). Specifically, an Dumbo-MVBA instance with identifier id proceeds as:

1. *Dispersal phase* (line 1-2, 13-18). The $n$ parties activate $n$ concurrent instances of the provable dispersal PD subprotocol. Each party $\mathcal{P}_i$ is the designated sender of a particular PD instance $\mathsf{PD}[\langle \mathsf{id}, i \rangle]$, through which $\mathcal{P}_i$ can disperse the coded fragments of its input $v_i$ across the network.
2. *Finish phase* (line 3, 19-35). This has a three-round structure to allow all parties consistently quit PD instances. It begins when a sender produces the *done* proof for its PD instance and multicasts *done* to all parties through a Done message, and finishes when all parties receive a Finish message attesting that at least $2f + 1$ PD instances has been "done". In addition, once receiving valid Finish, a party invokes *abandon*() to explicitly quit from all PD instances.
3. *Elect-ID phase* (line 5). Then all parties invoke the coin scheme Election, such that they obtain a common pseudo-random number $l$ over $[n]$. The common coin $l$ represents the identifier of a pair of $(\mathsf{PD}[\langle \mathsf{id}, l \rangle], \mathsf{RC}[\langle \mathsf{id}, l \rangle])$ instances.
4. *Recast-vote phase* (line 6-9, 36-39). Upon obtaining the coin $l$, the parties attempt to agree on whether to invoke the $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ instance or not. This phase has to cope with a major limit of RC subprotocol, that the $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ instance requires all parties to invoke it to reconstruct a common value. To this end, the *recast-vote* phase is made of a two-step structure. First, each party multicasts its locally recorded $lock[l]$ through RcBallotPrepare message, if the $\mathsf{PD}[\langle \mathsf{id}, l \rangle]$ instance actually delivers $lock[l]$; otherwise, it multicasts $\perp$ through RcBallotPrepare message. Then, each party waits for up to $2f + 1$ RcBallotPrepare from distinct parties, if it sees valid $lock[l]$ in these messages, it immediately activates $\mathsf{ABA}[\langle \mathsf{id}, l \rangle]$ with input 1, otherwise, it invokes $\mathsf{ABA}[\langle \mathsf{id}, l \rangle]$ with input 0. The above design follows the idea of biased validated binary agreement presented by Cachin et al. in [10], and $\mathsf{ABA}[\langle \mathsf{id}, l \rangle]$ must return 1 to each party, when $f + 1$ honest parties enter the phase with valid $lock[l]$.
5. *Recast phase* (line 10-12). When $\mathsf{ABA}[\langle \mathsf{id}, l \rangle]$ returns 1, all honest parties would enter this phase and there is always at least one honest party has delivered the valid *lock* regarding $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$. As such, the parties can always invoke the corresponding $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ instance to reconstruct a common value $v_l$. In case the recast value $v_l$ does not satisfy the external predicate, the parties can consistently go back to *elect-ID phase*, which is trivial because all parties have the same external predicate; otherwise, they output $v_l$.

### 6.3 Analyses of Dumbo-MVBA

Here we present the detailed proofs along with the complexity analyses for our Dumbo-MVBA construction.

**Security intuition**. The Dumbo-MVBA protocol described by Algorithm 4 solves asynchronous validate byzantine agreement among $n$ parties against adaptive adversary controlling $f \leq \lfloor \frac{n-1}{3} \rfloor$ parties, given (i) adaptively secure $f$-resilient APDB protocol, (ii) adaptively secure $f$-resilient ABA protocol, (iii) adaptively secure $(f + 1, n)$-Coin protocol (in the random oracle model), and (iv) adaptively secure $(f + 1, n)$ threshold signatures. We highlight here the key intuitions as follows:

- Termination and safety of *finish phase*. If any honest party leaves the finish phase and enters the elect-ID phase, then: (i) all honest parties will leave the finish phase, and (ii) at least $2f+1$ parties have produced *done* proofs for their dispersals.
- Termination and safety of *elect-ID phase*. Since the threshold of Election is $2f + 1$, $\mathcal{A}$ cannot learn which dispersals are elected to recover before $f + 1$ honest parties explicitly abandon all

**Algorithm 4** Dumbo-MVBA protocol with identifier id and external Predicate, for each party $\mathcal{P}_i$

**let** $provens \leftarrow 0$, $RDY \leftarrow \{\ \}$
**for** each $j \in [n]$ **do**
  **let** $store[j] \leftarrow \emptyset$, $lock[j] \leftarrow \emptyset$, $rc\text{-}ballot[j] \leftarrow 0$
  **initialize** a provable dispersal instance $\mathsf{PD}[\langle \mathsf{id}, j \rangle]$

1: **upon** receiving input $v_i$ s.t. $\mathsf{Predicate}(v_i) = true$ **do**
2:   pass $v_i$ into $\mathsf{PD}[\langle \mathsf{id}, i \rangle]$ as input ▷ dispersal phase
3:   **wait** for receiving any valid FINISH message ▷ finish phase
4:   **for** each $k \in \{1, 2, 3, \dots\}$ **do**
5:     $l \leftarrow \mathsf{Election}[\langle \mathsf{id}, k \rangle]$ ▷ elect-id phase
6:     **if** $lock[l] \neq \emptyset$ **then multicast** $(\text{RcBallotPrepare}, \mathsf{id}, l, lock)$ ▷ recast-vote phase
7:     **else multicast** $(\text{RcBallotPrepare}, \mathsf{id}, l, \perp)$
8:     **wait** for $rc\text{-}ballot[l] = 1$ or receiving $2f+1$ $(\text{RcBallotPrepare}, \mathsf{id}, l, \cdot)$ messages from distinct parties
9:       $b \leftarrow \mathsf{ABA}[\langle \mathsf{id}, l \rangle](rc\text{-}ballot[l])$
10:       **if** $b = 1$ **then** ▷ recast phase
11:         $v_l \leftarrow \mathsf{RC}[\langle \mathsf{id}, l \rangle](store[l], lock[l])$
12:         **if** $\mathsf{Predicate}(v_l) = true$ **then** output $v_l$

13: **upon** $\mathsf{PD}[\langle \mathsf{id}, j \rangle]$ delivers $store$ **do** ▷ record $store[j]$ for each $\mathsf{PD}[\langle \mathsf{id}, j \rangle]$
14:   $store[j] \leftarrow store$

15: **upon** $\mathsf{PD}[\langle \mathsf{id}, j \rangle]$ delivers $lock$ **do** ▷ record $lock[j]$ for each $\mathsf{PD}[\langle \mathsf{id}, j \rangle]$
16:   $lock[j] \leftarrow lock$

17: **upon** $\mathsf{PD}[\langle \mathsf{id}, i \rangle]$ delivers $done$ **do** ▷ multicast completeness proof $done$ for $\mathsf{PD}[\langle \mathsf{id}, i \rangle]$
18:   **multicast** $(\text{Done}, \mathsf{id}, done)$

19: **upon** receiving $(\text{Done}, \mathsf{id}, done)$ from party $\mathcal{P}_j$ for the first time **do**
20:   **if** $\mathsf{ValidateDone}(\langle \mathsf{id}, j \rangle, done) = 1$ **then**
21:     $provens \leftarrow provens + 1$
22:     **if** $provens = 2f + 1$ **then** ▷ one honest READY $\Rightarrow 2f+1$ DONE $\Rightarrow f+1$ honest DONE
23:       $\rho_{rdy,i} \leftarrow \mathsf{SignShare}_{(f+1)}(sk_i, \langle \text{Ready}, \mathsf{id} \rangle)$
24:       **multicast** $(\text{Ready}, \mathsf{id}, \rho_{rdy,i})$

25: **upon** receiving $(\text{Ready}, \mathsf{id}, \rho_{rdy,j})$ from party $\mathcal{P}_j$ for the first time **do**
26:   **if** $\mathsf{VerifyShare}_{(f+1)}(\langle \text{Ready}, \mathsf{id} \rangle, (j, \rho_{rdy,j})) = 1$ **then**
27:     $RDY \leftarrow RDY \cup (j, \rho_{rdy,j})$
28:     **if** $|RDY| = f + 1$ **then** ▷ $f+1$ READY $\Rightarrow$ one honest READY
29:       $\sigma_{rdy} \leftarrow \mathsf{Combine}_{(f+1)}(\langle \text{Ready}, \mathsf{id} \rangle, RDY)$
30:       **multicast** $(\text{Finish}, \mathsf{id}, \sigma_{rdy})$ to all, if was not sent before

31: **upon** receiving $(\text{Finish}, \mathsf{id}, \sigma_{rdy})$ from party $\mathcal{P}_j$ for the first time **do** ▷ valid FINISH $\Rightarrow f+1$ READY
32:   **if** $\mathsf{VerifyThld}_{(f+1)}(\langle \text{Ready}, \mathsf{id} \rangle, \sigma_{rdy}) = 1$ **then**
33:     $abandon(\langle \mathsf{id}, j \rangle)$ for each $j \in [n]$
34:     **multicast** $(\text{Finish}, \mathsf{id}, \sigma_{rdy})$ to all, if was not sent before
35:   **else discard** this invalid message

36: **upon** receiving $(\text{RcBallotPrepare}, \mathsf{id}, l, lock)$ from $\mathcal{P}_j$ **do**
37:   **if** $\mathsf{ValidateLock}(\langle \mathsf{id}, l \rangle, lock) = 1$ **then**
38:     $lock[l] \leftarrow lock$
39:     $rc\text{-}ballot[l] \leftarrow 1$ ▷ $rc\text{-}ballot[l] = 1 \Rightarrow lock[l]$ is valid $\Rightarrow \mathsf{PD}[\langle \mathsf{id}, j \rangle]$ is recoverable

dispersals, which prevents the adaptive adversary from "tampering" the values dispersed by uncorrupted parties. Moreover, $\mathsf{Election}$ terminates in constant time.
- Termination and safety of *recast-vote* and *recast*. The honest parties would consistently obtain either 0 or 1 from recast-vote. If recast-vote returns 1, all parties invoke a $\mathsf{RC}$ instance to recast the elected dispersal, which will recast a common value to all parties. Those cost expected constant time.
- Quality of *recast-vote* and *recast*. The probability that *recast-vote* returns 1 is at least 2/3. Moreover, conditioned on *recast-vote* returns 1, the probability that the *recast* phase returns an externally valid value is at least 1/2.

**The proofs of Dumbo-MVBA**. Now we prove our Algorithm 4 satisfies all properties of $\mathsf{MVBA}$ with all but negligible probability.

**Lemma 6.** *Suppose a party $\mathcal{P}_l$ multicasts* $(\textsc{Done}, \mathsf{id}, done)$*, where* $\mathsf{ValidateDone}(\langle \mathsf{id}, l\rangle, done) = 1$*. If all honest parties participate in the* $\mathsf{ABA}[\langle \mathsf{id}, l\rangle]$ *instance, then the* $\mathsf{ABA}[\langle \mathsf{id}, l\rangle]$ *returns 1 to all.*

*Proof.* If a party $\mathcal{P}_l$ did multicast a valid $(\textsc{Done}, \mathsf{id}, done)$, we know at least $f+1$ honest parties delivers valid $lock[l]$ s.t. $\mathsf{ValidateLock}(\mathsf{ID}, lock[l]) = 1$, due to the *Provability* properties of $\mathsf{APDB}$. Then according to the pseudocode of Algorithm 4, at least $f+1$ honest parties will multicast valid $(\textsc{RcBallotPrepare}, \mathsf{id}, l, lock)$ to all. In this case, since all honest parties need to wait for $2f+1$ RcBallotPrepare messages from distinct parties, then all honest parties must see a valid $(\textsc{RcBallotPrepare}, \mathsf{id}, l, lock)$ message. Therefore, all honest parties would input 1 to $\mathsf{ABA}[\langle \mathsf{id}, l\rangle]$ instance. From the *validity* properties of the $\mathsf{ABA}$ protocol, we know that the $\mathsf{ABA}[\langle \mathsf{id}, l\rangle]$ returns 1 to all.

**Lemma 7.** *Suppose all honest parties participate the* $\mathsf{ABA}[\langle \mathsf{id}, l\rangle]$ *instance and* $\mathsf{ABA}[\langle \mathsf{id}, l\rangle]$ *return 1 to all. If all honest parties invoke* $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$*, then the* $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$ *will return a same value to all honest parties. Besides, if $\mathcal{P}_l$ (sender) is an honest party, then the* $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$ *will return an externally validated value.*

*Proof.* Since the $\mathsf{ABA}[\langle \mathsf{id}, l\rangle]$ returns 1, we know at least one honest party inputs 1 to $\mathsf{ABA}$, due to the *validity* properties of $\mathsf{ABA}$. It also means that at least one honest party $\mathcal{P}_i$ receives a message $(\textsc{RcBallotPrepare}, \mathsf{id}, l, lock)$ which satisfies $\mathsf{ValidateLock}(\langle \mathsf{id}, l\rangle, lock) = 1$. According to the *Recast-ability* properties of $\mathsf{APDB}$, all honest parties will terminate in $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$, and recover a common value, conditioned on all honest parties invoke $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$.

In addition, if $\mathcal{P}_l$ is an honest party, $\mathcal{P}_l$ always inputs an externally valid value to $\mathsf{PD}[\langle \mathsf{id}, l\rangle]$, due to the *Recast-ability* properties of $\mathsf{APDB}$, the $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$ will return the exactly same valid value to all parties.

**Lemma 8.** *If an honest party invokes* $\mathsf{Election}[\langle \mathsf{id}, k\rangle]$*, then at least $2f+1$ distinct* $\mathsf{PD}$ *instances have completed, and all honest parties also invoked* $\mathsf{Election}[\langle \mathsf{id}, k\rangle]$*.*

*Proof.* Suppose an honest party $\mathcal{P}_i$ invokes $\mathsf{Election}[\langle \mathsf{id}, k\rangle]$, then it means that $\mathcal{P}_i$ receives a valid Finish message. It also means that at least $f+1$ parties multicast valid Ready message, it implies that at least one honest party received $2f+1$ valid Done messages from distinct parties. Since each Done message can verify the $\mathsf{PD}$ instance is indeed completed, at least $2f+1$ distinct $\mathsf{PD}$ instances have been completed.

In addition, for $k=1$, before an honest party invokes $\mathsf{Election}[\langle \mathsf{id}, 1\rangle]$, it must multicast the valid Finish message, if it was not sent before. For the other honest parties, they will also invoke the $\mathsf{Election}[\langle \mathsf{id}, 1\rangle]$, upon receiving a valid Finish message. For $k>1$, without loss of generality, suppose an honest party $\mathcal{P}_i$ halts after invoking $\mathsf{Election}[\langle \mathsf{id}, k\rangle]$, and another honest party $\mathcal{P}_j$ halts after invoking $\mathsf{Election}[\langle \mathsf{id}, k'\rangle]$, where $k' > k$. However, according the *agreement* of $\mathsf{ABA}$, all honest parties will output the same bit 0 (not recast) or 1 (to recast); in addition, according to the *recast-ability* properties of $\mathsf{APDB}$, all honest parties will recover the same value if $\mathsf{ABA}$ returns 1. So, if $\mathcal{P}_i$ halts after invoking $\mathsf{Election}[\langle \mathsf{id}, k\rangle]$, $\mathcal{P}_j$ shall also halt after invoking $\mathsf{Election}[\langle \mathsf{id}, k\rangle]$. Hence, the honest party $\mathcal{P}_j$ would not enter $\mathsf{Election}[\langle \mathsf{id}, k'\rangle]$, when another honest party $\mathcal{P}_i$ would not invoke $\mathsf{Election}[\langle \mathsf{id}, k'\rangle]$.

**Lemma 9. Termination**. *If every honest party $\mathcal{P}_i$ activates the protocol on identification* id *with proposing an input value $v_i$ such that* Predicate$(v_i) = true$*, then every honest party outputs a value $v$ for* id *in constant time.*

*Proof.* According to Algorithm 4, the Dumbo-MVBA protocol first executes $n$ concurrent PD instances. Since all honest parties start with externally valid values and all massages sent among honest parties have been delivered, from the *termination* of APDB, if no honest party abandons the PD, any honest parties can know at least $n - f$ PD instances have completed; if any honest party abandons the PD instances, it means that this party has seen a valid FINISH messages, which attests at least $n - f$ PD instances have completed.

When an honest party learns at least $n - f$ PD instances have completed, it will invoke Election$[\langle \mathsf{id}, k\rangle]$ to elect a random number $l$. From Lemma 8, we know all other honest parties also will invoke Election$[\langle \mathsf{id}, k\rangle]$. In addition, all honest parties will input a value to ABA$[\langle \mathsf{id}, l\rangle]$, from the *termination* and *agreement* properties of ABA, the ABA$[\langle \mathsf{id}, l\rangle]$ will return a same value to all. Next, let us consider three following cases:

- **Case 1:** ABA$[\langle \mathsf{id}, l\rangle]$ returns 1 to all. According to the *recast-ability* properties of APDB, the RC$[\langle \mathsf{id}, l\rangle]$ instance will terminate and recover a same value to all. The recast value can be valid and satisfy the global Predicate, then this value will be decided as output by all parties.
- **Case 2:** ABA$[\langle \mathsf{id}, l\rangle]$ returns 1 to all. Due to the *recast-ability* of APDB, the RC$[\langle \mathsf{id}, l\rangle]$ instance will terminate and recover a same value to all. The value can be invalid due to the global external Predicate, the honest parties will repeat Election, until Case 1 occasionally happens.
- **Case 3:** If ABA$[\langle \mathsf{id}, l\rangle]$ returns 0 to all, then the honest parties will repeat Election, until Case 1 occasionally happens.

Now, we prove that the protocol terminates, after sequentially repeating ABA (and RC). Recall all honest parties start with dispersing externally valid values, so after Election$[\langle \mathsf{id}, k\rangle]$ returns $l$ for every $k \geq 1$, the probability that $\mathcal{P}_l$ is honest and completes PD$[\langle \mathsf{id}, l\rangle]$ is at least $p = 1/3$. Due to the *unbiasedness* of Election, the coin $\ell$ returned by Election is uniform over $[n]$.

As such, let the event $E_k$ represent that the protocol does not terminate when Election$[\langle \mathsf{id}, k\rangle]$ has been invoked, so the probability of the event $E_k$, $\Pr[E_k] \leq (1-p)^k$. It is clear to see $\Pr[E_k] \leq (1-p)^k \to 0$ when $k \to \infty$, so the protocol eventually halts. Moreover, let $K$ to be the random variable that the protocol just terminates when $k = K$, so $\mathbb{E}[K] \leq \sum_{K=1}^{\infty} K(1-p)^{K-1}p = 1/p = 3$, indicating the protocol terminates in expected constant time.

**Lemma 10. External-Validity**. *If an honest party outputs a value $v$ for* id*, then* Predicate$(v) =$ *true.*

*Proof.* According to Algorithm 4, when an honest party outputs a value $v$, there is always Predicate$(v) =$ *true*. Therefore, the external-validity trivially holds.

**Lemma 11. Agreement**. *If any two honest parties output $v$ and $v'$ for* id *respectively, then $v = v'$.*

*Proof.* From lemma 8, we know if an honest party invokes Election$[\langle \mathsf{id}, k\rangle]$, then all honest parties also invoke Election$[\langle \mathsf{id}, k\rangle]$. From the *agreement* properties of Election, all honest parties get the same coin $l$. Hence, all honest parties will participate in the same ABA$[\langle \mathsf{id}, l\rangle]$ instance. Besides, due to the *agreement* of ABA, all honest parties will get a same bit $b$. Hence, upon ABA$[\langle \mathsf{id}, l\rangle] = 1$, then all honest parties will participate in the same RC$[\langle \mathsf{id}, l\rangle]$ instance. According to the *recast-ability* property of APDB, all honest parties must output the same value.

**Lemma 12. Quality**. *If an honest party outputs $v$ for* id*, the probability that $v$ was proposed by the adversary is at most 1/2.*

*Proof.* Due to Lemma 8, as long as an honest party activates Election, at least $2f + 1$ distinct PD instances have completed, which means these PD instances' senders can produce valid completeness *done* proofs. Moreover, if any honest party invokes Election$[\langle \mathsf{id}, k\rangle]$, all honest parties will eventually invoke Election$[\langle \mathsf{id}, k\rangle]$ as well. Suppose Election$[\langle \mathsf{id}, k\rangle]$ returns $l$, then all honest parties

will participate in the ABA[$\langle$id$, l\rangle$] instance. If the sender $\mathcal{P}_l$ has completed the PD protocol, due to Lemma 6, the ABA[$\langle$id$, l\rangle$] will return 1 to all.

Then, if ABA[$\langle$id$, l\rangle$] returns 0, all parties will go to the next iteration to enter Election[$\langle$id$, k+1\rangle$]; otherwise, ABA[$\langle$id$, l\rangle$] returns 1, all honest parties will participate in the RC[$\langle$id$, l\rangle$] instance, and the RC[$\langle$id$, l\rangle$] instance will return a common value to all parties, due to Lemma 7.

Let $\mathbb{P}_a$ to denote the set of the parties that are already corrupted by the adversary, when the adversary can tell the output of Election with non-negligible probability. Due to the *unpredictability* property of Election, upon the adversary can realize the output of Election, at least $f+1$ honest parties have already activated Election and therefore have abandoned all PD instances. This further implies that, once the adversary realizes the output of Election, the adversary can no longer disperse adversarial values by adaptively corrupting any so-far-uncorrupted senders outside $\mathbb{P}_a$.

Moreover, when the adversary is able to predicate the output of Election, at least $2f+1$ PD instances have been completed, out of which at most $|\mathbb{P}_a|$ instances are dispersed by the adversary. Therefore, we consider the worst case that: (i) only $f+1$ honest parties have completed their PD instances, and (ii) $|\mathbb{P}_a| = f$ and these $f$ PD instances sent by the adversary have completed. In addition, due to the *unbiasedness* property of Election, the adversary cannot bias the distribution of the output of Election. So Election[$\langle$id$, k\rangle$] returns a coin $l$ that is uniformly sampled over $[n]$, which yields the next three cases for any $k \in \{1, 2, \dots\}$:

- **Case 1:** If the sender $\mathcal{P}_l$ has not completed the PD instance yet, and the ABA[$\langle$id$, l\rangle$] returns 0, then repeats Election, the probability of this case at most is 1/3; in such the case, the protocol would go to Election to repeat;
- **Case 2:** If the sender $\mathcal{P}_l$ has completed the PD protocol and the sender' input was determined by the adversary (which might or might not be valid regarding the global predicate), the probability of this case at most is 1/3;
- **Case 3:** If the sender $\mathcal{P}_l$ has completed the PD protocol and the sender' input was not determined by the adversary, the probability of this case at least is 1/3;

Hence, the probability of deciding an output value $v$ proposed by the adversary is at most $\sum_{k=1}^{\infty}(1/3)^k = 1/2$.

**Theorem 3.** *In random oracle model, the protocol described by algorithm 4 solves asynchronous validate byzantine agreement (*Dumbo-MVBA*) among $n$ parties against adaptive adversary controlling $f < n/3$ parties, given (i) $f$-resilient* APDB *protocol against adaptive adversary, (ii) $f$-resilient* ABA *protocol against adaptive adversary, and (iii) adaptively secure non-interactive $(2f+1, n)$ and $(f+1, n)$ threshold signatures.*

*Proof.* Lemma 9, 10, 11 and 12 complete the proof.

**The complexity analysis of Dumbo-MVBA**. The Dumbo-MVBA achieves: (i) asymptotically optimal round and message complexities, and (ii) asymptotically optimal communicated bits $\mathcal{O}(\ell n + \lambda n^2)$ for any input $\ell \geq \lambda n$.

According to the pseudocode of algorithm 4, the breakdown of its cost can be briefly summarized in the next five phases: (i) the dispersal phase that consists of the $n$ concurrent PD instances; (ii) the finish phase which is made of three all-to-all multicasts of Done, Ready and Finish messages; (iii) the elect-ID phase where is an invocation of Election; (iv) the recast-vote phase that has one all-to-all multicast of RcBallotPrepare messages and an invocation of ABA instance; (v) the recast phase where is to executes an RC instance.

Due to the complexity analysis of APDB in section 5.3, we know the PD's message complexity is $\mathcal{O}(n)$ and its communication complexity is $\mathcal{O}(\ell + n\lambda)$; the RC's message complexity is $\mathcal{O}(n^2)$ and its communication complexity is $\mathcal{O}(n\ell + n^2\lambda)$. So the complexities of Dumbo-MVBA protocol can be summarized as:

- **Running time:** The protocol terminates in expected constant running time due to Lemma 9.
- **Message complexity:** In the dispersal phase, there are $n$ PD instances, each of which incurs $\mathcal{O}(n)$ messages. In the finish phase, there are three all-to-all multicasts, which costs $\mathcal{O}(n^2)$

messages. In the elect phase, there is one common coin, that incurs $\mathcal{O}(n^2)$ messages. In the rc-vote phase, there is one all-to-all multicast and one ABA instance, which incurs $\mathcal{O}(n^2)$ messages. In recast phase, there is only one RC instance, thus yielding $\mathcal{O}(n^2)$ messages. Moreover, the elect phase, the rc-vote-phase, and the recast phase would be repeated for expected 3 times. To sum up, the overall message complexity of the Dumbo-MVBA protocol is $\mathcal{O}(n^2)$.

- **Communication complexity:** In the dispersal phase, there are $n$ PD instances, each of which incurs $\mathcal{O}(\ell + n\lambda)$ bits. In the finish phase, there are three all-to-all multicasts, which corresponds to $\mathcal{O}(n^2)$ $\lambda$-bit messages. In the elect-ID phase, there is one common coin, that incurs $\mathcal{O}(n^2\lambda)$ bits. In the recast-vote phase, there is one all-to-all multicast and one ABA instance, thus incurring $\mathcal{O}(n^2)$ messages, each of which contains at most $\lambda$ bits. In the recast phase, there is only one RC instance, thus yielding $\mathcal{O}(n^2)$ messages, each of which contains at most $\mathcal{O}(\ell + n\lambda)$ bits. Moreover, the elect phase and the rc-vote phase would be repeated for expected 3 times, and the recast phase would be repeated for expected 2 times. Hence, the communication complexity of the Dumbo-MVBA protocol is $\mathcal{O}(n\ell + n^2\lambda)$.

Note if considering $\ell \geq \mathcal{O}(n\lambda)$, the Dumbo-MVBA protocol realizes optimal communication complexity $\mathcal{O}(n\ell)$.

## 7 A generic optimal MVBA framework

The *dispersal-then-recast* methodology can also be applied to bootstrap any existing MVBA to realize optimal communication for sufficiently large input. We call this extension protocol Dumbo-MVBA⋆. The key idea is to invoke the underlying MVBA with taking as input the small-size proofs of APDB. Though Dumbo-MVBA⋆ is a "reduction" from MVBA to MVBA itself, an advanced module instead of more basic building block such as binary agreement, this self-bootstrap technique can better utilize MVBA to achieve a simple modular design as explained in Figure 2, and we note it does not require the full power of APDB (and thus can potentially remove the rounds of communication generating the *done* proof).

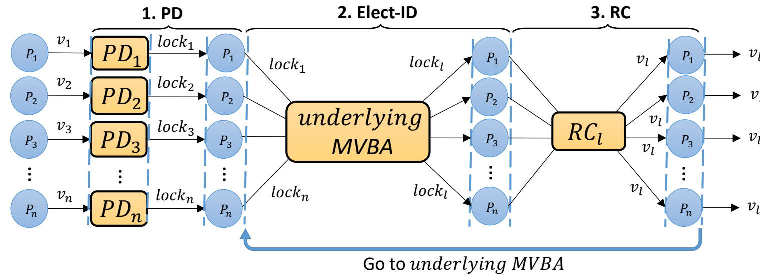


**Fig. 2.** The execution flow of Dumbo-MVBA⋆.

### 7.1 Overview of Dumbo-MVBA⋆ framework

As shown in Figure 2, the generic framework still follows the idea of *dispersal-then-recast*: (i) each party disperses its own input value and obtains a *lock* proof attesting the recast-ability of its own dispersal; (ii) then, the parties can invoke any existing MVBA as a black-box to "elect" a valid *lock* proof, and then recover the already-dispersed value, until all parties recast and decide an externally valid value.

This generic Dumbo-MVBA⋆ framework presents a simple modular design that can enhance any existing MVBA protocol to achieve optimal communication for sufficiently large input, without scarifying the message complexity and running time of underlying MVBA. In particular, when instantiating the framework with using the MVBA protocol due to Abraham et al. [2], we can obtain an optimal MVBA protocol that outperforms the state-of-the-art, since it achieves only $\mathcal{O}(n\ell+n^2\lambda)$ communicated bits, without giving up the optimal running time and message complexity.

**Algorithm 5** The Dumbo-MVBA$\star$ protocol with identification id and external Predicate(), for each party $\mathcal{P}_i$

**let** $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k\rangle]$ to be a MVBA instance which takes as input $\langle i, lock_i\rangle$ and is parameterized by the next external predicate:

$$\mathsf{Predicate_{Election}}(\mathsf{id}, i, lock_i) \equiv (\mathsf{ValidateLock}(\langle \mathsf{id}, i\rangle, lock_i) \ \wedge \ i \in [n])$$

**for** each $j \in [n]$ **do**
  **let** $store[j] \leftarrow \emptyset$ and **initialize** an instance $\mathsf{PD}[\langle \mathsf{id}, j\rangle]$

1: **upon** receiving input $v_i$ s.t. $\mathsf{Predicate}(v_i) = 1$ **do**
2:   pass $v_i$ into $\mathsf{PD}[\langle \mathsf{id}, i\rangle]$ as input ▷ provable dispersal phase
3:   **wait** for $\mathsf{PD}[\langle \mathsf{id}, i\rangle]$ delivers $lock_i$
4:     **for** each $k \in \{1, 2, 3, \dots\}$ **do**
5:       $\langle l, lock_l\rangle \leftarrow \mathsf{MVBA}_{under}[\langle \mathsf{id}, k\rangle](\langle i, lock_i\rangle)$ ▷ elect a finished dispersal to recast
6:       $v_l \leftarrow \mathsf{RC}[\langle \mathsf{id}, l\rangle](store[l], lock_l)$
7:       **if** $\mathsf{Predicate}(v_l) = true$ **then** output $v_l$

8: **upon** $\mathsf{PD}[\langle \mathsf{id}, j\rangle]$ delivers $store$ **do** ▷ record $store[j]$ for each $\mathsf{PD}[\langle \mathsf{id}, j\rangle]$
9:   $store[j] \leftarrow store$

## 7.2 Details of the Dumbo-MVBA$\star$ protocol

Here is our generic Dumbo-MVBA$\star$ framework. Informally, a Dumbo-MVBA$\star$ instance with identification id (as illustrated in Algorithm 5) proceeds as:

1. *Dispersal phase* (line 1-3, 8-9). $n$ concurrent PD instances are activated. Each party $\mathcal{P}_i$ is the designated sender of the instance $\mathsf{PD}[\langle \mathsf{id}, i\rangle]$, through which $\mathcal{P}_i$ disperses its input's fragments across the network.
2. *Elect-ID phase* (line 4-5). As soon as the party $\mathcal{P}_i$ delivers $lock_i$ during its dispersal instance $\mathsf{PD}[\langle \mathsf{id}, i\rangle]$, it takes the proof $lock_i$ as input to invoke a concrete MVBA instance with identifier $\langle \mathsf{id}, k\rangle$, where $k \in \{1, 2, \dots\}$. The external validity of underlying MVBA instance is specified to output a valid $lock_l$ for any PD instance $\mathsf{PD}[\langle \mathsf{id}, l\rangle]$.
3. *Recast phase* (line 6-7). Eventually, the $\mathsf{MVBA}[\langle \mathsf{id}, k\rangle]$ instance returns to all parties a common $lock_l$ proof for the $\mathsf{PD}[\langle \mathsf{id}, l\rangle]$ instance, namely, MVBA elects a party $\mathcal{P}_l$ to recover its dispersal. Then, all honest parties invoke $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$ to recover a common value $v_l$. If the recast $v_l$ is not valid, every party $\mathcal{P}_i$ can realize locally due to the same global Predicate, so each $\mathcal{P}_i$ can consistently go back the *elect-ID phase* to repeat the election by running another $\mathsf{MVBA}[\langle \mathsf{id}, k+1\rangle]$ instance with still passing $lock_i$ as input, until a valid $v_l$ can be recovered by an elected $\mathsf{RC}[\langle \mathsf{id}, l\rangle]$ instance.

## 7.3 Analyses of Dumbo-MVBA$\star$

Here we present the detailed proofs along with the complexity analyses for our Dumbo-MVBA$\star$ construction.

**Security intuition**. The Dumbo-MVBA$\star$ protocol described by Algorithm 5 realizes (optimal) MVBA among $n$ parties against adaptive adversary controlling $f \leq \lfloor \frac{n-1}{3} \rfloor$ parties, given (i) $f$-resilient APDB protocol against adaptive adversary (with all properties but abandon-ability and provability), (ii) adaptively secure $f$-resilient MVBA protocol. The key intuitions of Dumbo-MVBA$\star$ as follows:

- The repetition of the phase (2) and the phase (3) can terminate in expected constant time, as the quality of every underlying MVBA instance ensures that there is at least 1/2 probability of electing a PD instance whose sender was not corrupted before invoking MVBA.
- As such, the probability of not recovering any externally valid value to halt exponentially decreases with the repetition of *elect-ID* and *recast*. Hence only few (i.e., two) underlying MVBA instances and RC instances will be executed on average.

**The proofs of Dumbo-MVBA$\star$.** Now we prove that Algorithm 5 satisfies all properties of MVBA except with negligible probability.

**Lemma 13.** *Suppose a party $\mathcal{P}_i$ delivers $\langle l, lock_l \rangle$ in any $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ that $k \in [n]$, then all honest parties would invoke $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ and recover a common value from $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$. Besides, if $\mathcal{P}_l$ (i.e., the sender of $\mathsf{PD}[\langle \mathsf{id}, l \rangle]$) was not corrupted before $lock_l$ was delivered, then the $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ returns a validated value.*

*Proof.* If any honest party delivers $\langle l, lock_l \rangle$ in any $\mathsf{MVBA}_{under}$ instance, all honest parties deliver the same $\langle l, lock_l \rangle$ in this $\mathsf{MVBA}_{under}$ instance, so all honest parties would invoke $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$. Moreover, due to the specification of $\mathsf{Predicate}_{\mathsf{Election}}$ shown in Algorithm 5, all honest parties deliver $\langle l, lock_l \rangle$, s.t. $\mathsf{ValidateLock}(\langle \mathsf{id}, l \rangle, lock_l) = 1$. According to the *recast-ability* property of APDB, all honest parties (that invoke $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$) will terminate and output the same value (or the same $\perp$). In addition, the *recast-ability* property also states: conditioned on that $\mathcal{P}_l$ was not corrupted before delivering $lock_l$ and it took as input a valid value $v_l$ to disperse in $\mathsf{PD}[\langle \mathsf{id}, l \rangle]$, the $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ will return to all parties the valid value $v_l$.

**Lemma 14. Termination**. *If every honest party $\mathcal{P}_i$ activates the protocol on identification* id *with proposing an input value $v_i$ such that $\mathsf{Predicate}(v_i) = true$, then every honest party outputs a value $v$ for* id*. Moreover, if the expected running time of the underlying $\mathsf{MVBA}_{under}$ is $\mathcal{O}(\mathsf{poly}_{rt}(n))$, Dumbo-MVBA$\star$ is expected to run in $\mathcal{O}(\mathsf{poly}_{rt}(n))$ time.*

*Proof.* According to Algorithm 5, Dumbo-MVBA$\star$ firstly executes $n$ concurrent PD instance. From the *termination* of APDB: if a sender $\mathcal{P}_s$ is honest and all honest parties activate $\mathsf{PD}[\langle \mathsf{id}, s \rangle]$ without abandoning, then the honest sender $\mathcal{P}_s$ can deliver $lock_s$ for identification $\langle \mathsf{id}, s \rangle$ s.t. $\mathsf{ValidateLock}(\langle \mathsf{id}, s \rangle, lock_s) = 1$.

In case every honest party $\mathcal{P}_i$ passes an input to its PD instance, all honest parties can deliver a lock proof *lock* from PD, which satisfies the $\mathsf{Predicate}_{\mathsf{Election}}$ of $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$. Hence, each honest party $\mathcal{P}_i$ will pass a valid $\langle i, lock_i \rangle$ as input into $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ for each iteration $k \in [n]$. Following the *agreement* and *termination* of MVBA, all honest parties can get the same output $\langle l, lock_l \rangle$ from each $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ instance.

Due to the *external-validity* of the underlying MVBA, the output $\langle l, lock_l \rangle$ of each $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ shall satisfy $\mathsf{Predicate}_{\mathsf{Election}}(\mathsf{id}, l, lock_l) = 1$. After $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ returns $\langle l, lock_l \rangle$, the $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ will be invoked and return a same value $v_l$ to all in constant time due to Lemma 13. Let us consider two cases for any $k \in \{1, 2, \dots\}$ as follows:

- **Case 1:** If the value $v_l$ returned by $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ is valid, then output the value.
- **Case 2:** If the value $v_l$ returned by $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$ is not valid, the parties will go back to the elect-ID phase to execute $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k+1 \rangle]$, until a valid value will be decided.

Now, we prove that the honest parties would terminate in expected constant time, except with negligible probability. Due to the *quality* properties of the MVBA, the probability that $\langle l, lock_l \rangle$ was proposed by the adversary is at most $1/2$ for each $\mathsf{MVBA}_{under}$ instance with different identification $\langle \mathsf{id}, k \rangle$. In addition, due to the *recast-ability* of APDB, whenever $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$'s output $\langle l, lock_l \rangle$ was not proposed by the adversary, a valid value can be collectively recovered by all honest parties due to $\mathsf{RC}[\langle \mathsf{id}, l \rangle]$. So the probability that an externally valid $v_l$ is recover after invoking each $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ is at least $p = 1/2$. Let the event $E_k$ represent that the protocol does not terminate when $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ has been invoked for $k$ times, so the probability of the event $E_k$, $\Pr[E_k] \leq (1-p)^k$. It is clear to see $\Pr[E_k] \leq (1-p)^k \to 0$ when $k \to \infty$, so the protocol eventually halts. Moreover, let $K$ to be the random variable that the protocol just terminates when $k = K$, so $\mathbb{E}[K] \leq \sum_{K=1}^{\infty} K(1-p)^{K-1}p = 1/p = 2$, indicating the protocol is expected to terminate after sequentially invoking $\mathsf{MVBA}_{under}[\langle \mathsf{id}, k \rangle]$ twice.

**Lemma 15. External-Validity**. *If an honest party outputs a value $v$ for* id*, then $\mathsf{Predicate}(v) = true$.*

*Proof.* According to Algorithm 5, when an honest party outputs a value, $\mathsf{Predicate}(v) = true$. Therefore, the external-validity trivially follows.

**Lemma 16. Agreement**. *If any two honest parties output $v$ and $v'$ for* id *respectively, then $v = v'$.*

*Proof.* From the *agreement* property of MVBA, all honest parties get the same output $\langle l, lock_l \rangle$. Hence, all honest parties will participate in the common RC[$\langle$id$, l\rangle$] instance. Moreover, due to the *recast-ability* property of APDB, all honest parties will recover the same value from each invoked RC[$\langle$id$, l\rangle$]. In addition, all honest parties have the same a-priori known predicate, and they output only when the recast value from RC[$\langle$id$, l\rangle$] satisfying this global predicate. Thus the decided output of any two honest parties must be the same.

**Lemma 17. Quality**. *If an honest party outputs $v$ for* id*, the probability that $v$ was proposed by the adversary is at most 1/2.*

*Proof.* Due to the *external-validity* and *agreement* properties of the underlying MVBA$_{under}$, every honest party can get the same output $\langle l, lock_l \rangle$ from MVBA$_{under}$[$\langle$id$, k\rangle$] which satisfies the *external* Predicate$_{\text{Election}}$, namely, $lock_i$ is the valid lock proof for the sender $\mathcal{P}_l$'s dispersal instance due to ValidateLock($\langle$id$, l\rangle, lock_l) = true$.

Then, all honest parties will participate in the same RC[$\langle$id$, l\rangle$] instance, according to Algorithm 5. From Lemma 13, we know the RC[$\langle$id$, l\rangle$] will terminate and output a common value $v_l$ to all. Because of the *quality* properties of the MVBA, the probability that $\langle l, lock_l \rangle$ was proposed by the adversary is at most 1/2. So RC[$\langle$id$, l\rangle$] returns a value $v_l$ that might correspond the next two cases:

- **Case 1:** The sender $\mathcal{P}_l$ was corrupted by $\mathcal{A}$ (before delivering $lock_l$);
- **Case 2:** The sender $\mathcal{P}_l$ was not corrupted by $\mathcal{A}$ (before delivering $lock_l$), and executing RC[$\langle$id$, l\rangle$] must output the valid value proposed by this sender (when it was not corrupted), due to the *recast-ability* of APDB;

Due to the *fairness* of underlying MVBA$_{under}$, the probability of Case 1 is at most 1/2, while the probability of Case 2 is at least 1/2, so the probability of deciding a value $v_l$ was proposed by the adversary is at most 1/2.

**Theorem 4.** *The protocol described by algorithm 5 (*Dumbo-MVBA$\star$*) solves asynchronous validate Byzantine agreement among $n$ parties against adaptive adversary controlling $f < n/3$ parties, given (i) $f$-resilient* APDB *protocol against adaptive adversary, and (ii) $f$-resilient* MVBA *protocol against adaptive adversary.*

*Proof.* Lemma 14, 15, 16, and 17 complete the proof.

**The complexity analysis of Dumbo-MVBA$\star$.** According to the pseudocode of Algorithm 5, the cost of Dumbo-MVBA$\star$ is incurred in the next three phase: (i) the dispersal phase consisting of $n$ concurrent PD instances; (ii) the elect-ID phase consisting of few expected constant number (i.e., two) of underlying MVBA instances; (iii) the recast phase consisting of few expected constant number (i.e., two) of RC instances.

Recall the complexities of PD and RC protocols: PD costs $\mathcal{O}(n)$ messages, $\mathcal{O}(\ell + n\lambda)$ bits, and $\mathcal{O}(1)$ running time; RC costs $\mathcal{O}(n^2)$ messages, $\mathcal{O}(n\ell + n^2\lambda)$ bits, and $\mathcal{O}(1)$ running time. Suppose the underlying MVBA module incurs expected $\mathcal{O}(\mathsf{poly}_{rt}(n))$ running time, expected $\mathcal{O}(\mathsf{poly}_{mc}(n))$ messages, and expected $\mathcal{O}(\mathsf{poly}_{cc}(\ell, \lambda, n))$ bits, where $\mathcal{O}(\mathsf{poly}_{mc}(n)) \geq \mathcal{O}(n^2)$ and $\mathcal{O}(\mathsf{poly}_{cc}(\ell, \lambda, n)) \geq \mathcal{O}(\ell n + n^2)$ due to the lower bounds of adaptively secure MVBA. Thus the complexities of Dumbo-MVBA$\star$ are:

- **Running time:** Since PD and RC are deterministic protocols with constant running timing, the running time of Dumbo-MVBA$\star$ is dominated by the underlying MVBA module, namely, $\mathcal{O}(\mathsf{poly}_{rt}(n))$.
- **Message complexity:** The message complexity of $n$ PD instances (or a RC instance) is $\mathcal{O}(n^2)$. The message complexity of the underlying MVBA is $\mathcal{O}(\mathsf{poly}_{mc}(n))$, where $\mathcal{O}(\mathsf{poly}_{mc}(n)) \geq \mathcal{O}(n^2)$. As such, the messages complexity of Dumbo-MVBA$\star$ is dominated by the underlying MVBA protocol, namely, $\mathcal{O}(\mathsf{poly}_{mc}(n))$.

- **Communication complexity:** The communication of $n$ concurrent $\mathsf{PD}$ instances (or a $\mathsf{RC}$ instance) is $\mathcal{O}(n\ell + n^2\lambda)$. The underlying $\mathsf{MVBA}$ module incurs $\mathcal{O}(\mathsf{poly}_{cc}(\lambda, \lambda, n))$ bits. So the overall communication complexity of Dumbo-MVBA⋆ is $\mathcal{O}(\ell n + \lambda n^2 + \mathsf{poly}_{cc}(\lambda, \lambda, n))$.

As shown in Table 2, Dumbo-MVBA⋆ reduces the communication of the underlying MVBA from $\mathcal{O}(\mathsf{poly}_{cc}(\ell, \lambda, n))$ to $\mathcal{O}(\ell n + \lambda n^2 + \mathsf{poly}_{cc}(\lambda, \lambda, n))$, which removes all superlinear terms factored by $\ell$ in the communication complexity. In particular, for sufficiently large input whose length $\ell \geq \max(\lambda n, \mathsf{poly}_{cc}(\lambda, \lambda, n)/n)$, Dumbo-MVBA⋆ coincides with the asymptotically *optimal* $\mathcal{O}(n\ell)$ communication.

**Table 2. Asymptotic performance of MVBA protocols for $\ell$-bit inputs**

| Protocols | Running time | Message Comp. | Comm. Comp. (bits) |
|---|---|---|---|
| underlying MVBA | $\mathcal{O}(\mathsf{poly}_{rt}(n))$ | $\mathcal{O}(\mathsf{poly}_{mc}(n))$ | $\mathcal{O}(\mathsf{poly}_{cc}(\ell, \lambda, n))$ |
| Dumbo-MVBA⋆ | $\mathcal{O}(\mathsf{poly}_{rt}(n))$ | $\mathcal{O}(\mathsf{poly}_{mc}(n))$ | $\mathcal{O}(\ell n + \lambda n^2 + \mathsf{poly}_{cc}(\lambda, \lambda, n))$ |

**Concrete instantiation**. Dumbo-MVBA⋆ can be instantiated by extending the MVBA protocol of Abraham et al. [2]. Moreover, the MVBA protocol of Abraham et al. achieved expected $\mathcal{O}(1)$ running time, $\mathcal{O}(n^2)$ messages and $\mathcal{O}(n^2\ell + n^2\lambda)$ bits, it's clear that our Dumbo-MVBA⋆ framework can extend their result to attain $\mathcal{O}(n\ell + n^2\lambda)$ bits without scarifying the optimal running time and message complexity.

Note if considering $\ell \geq \mathcal{O}(n\lambda)$, the above instantiation of Dumbo-MVBA⋆ realizes optimal $\mathcal{O}(n\ell)$ communication.

## 8 Conclusion

We present two MVBA protocols that reduce the communication cost of prior art [2,10] by an $\mathcal{O}(n)$ factor, where $n$ is the number of parties. These communication-efficient MVBA protocols also attain other optimal properties, asymptotically. Our results complement the recent breakthrough of Abraham et al. at PODC '19 [2] and solve the remaining part of the long-standing open problem from Cachin et al. at CRYPTO '01 [10].

Our MVBA protocols can immediately be applied to construct efficient asynchronous atomic broadcast with reduced communication blow-up as previously suggested in [10] and [31]. Moreover, they can provide better building blocks for the Dumbo BFT protocols [22], the recent constructions of practical asynchronous atomic broadcast that rely on MVBA at their heart for efficiency.

There are still a few interesting open problems left in the domain of MVBA, such as exploring various trade-offs to further reduce the communication and message complexities, e.g., by restricting the power of adversary and/or the number of corruptions.

## Acknowledgment

We would like to thank the anonymous reviewers of PODC '20 for their valuable comments and suggestions about this paper. Qiang Tang and Zhenliang Lu are supported in part by JD Digits via the JDD-NJIT-ISCAS Joint Blockchain Lab. Qiang is also supported in part by a Google Faculty Award.

## References

1. Abraham, I., Chan, T., Dolev, D., Nayak, K., Pass, R., Ren, L., Shi, E.: Communication complexity of byzantine agreement, revisited. In: Proc. ACM PODC 2019. pp. 317–326. ACM (2019)
2. Abraham, I., Malkhi, D., Spiegelman, A.: Asymptotically optimal validated asynchronous byzantine agreement. In: Proc. ACM PODC 2019. pp. 337–346. ACM (2019)

3. Ben-Or, M.: Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In: Proc. ACM PODC 1983. pp. 27–30. ACM (1983)
4. Ben-Or, M., El-Yaniv, R.: Resilient-optimal interactive consistency in constant time. Distributed Computing **16**(4), 249–262 (2003)
5. Ben-Or, M., Kelmer, B., Rabin, T.: Asynchronous secure computations with optimal resilience. In: Proc. ACM PODC 1994. pp. 183–192. ACM (1994)
6. Blahut, R.E.: Theory and practice of error control codes. Addison-Wesley (1983)
7. Boldyreva, A.: Threshold signatures, multisignatures and blind signatures based on the gap-diffie-hellman-group signature scheme. In: PKC 2003. pp. 31–46. Springer (2003)
8. Bracha, G.: Asynchronous byzantine agreement protocols. Information and Computation **75**(2), 130–143 (1987)
9. Cachin, C., Kursawe, K., Lysyanskaya, A., Strobl, R.: Asynchronous verifiable secret sharing and proactive cryptosystems. In: Proc. ACM CCS 2002. pp. 88–97 (2002)
10. Cachin, C., Kursawe, K., Petzold, F., Shoup, V.: Secure and efficient asynchronous broadcast protocols. In: Annual International Cryptology Conference. pp. 524–541. Springer (2001)
11. Cachin, C., Kursawe, K., Shoup, V.: Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. Journal of Cryptology **18**(3), 219–246 (2005)
12. Cachin, C., Tessaro, S.: Asynchronous verifiable information dispersal. In: Proc. IEEE SRDS 2005. pp. 191–201. IEEE (2005)
13. Cachin, C., Vukolic, M.: Blockchain consensus protocols in the wild (keynote talk). In: 31st International Symposium on Distributed Computing (DISC 2017). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2017)
14. Canetti, R., Rabin, T.: Fast asynchronous byzantine agreement with optimal resilience. In: Proc. ACM STOC 1993. pp. 42–51. ACM (1993)
15. Catalano, D., Fiore, D.: Vector commitments and their applications. In: PKC 2013
16. Correia, M., Neves, N.F., Veríssimo, P.: From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. The Computer Journal **49**(1), 82–96 (2006)
17. Dolev, D., Strong, H.R.: Authenticated algorithms for byzantine agreement. SIAM Journal on Computing **12**(4), 656–666 (1983)
18. Duan, S., Reiter, M.K., Zhang, H.: Beat: Asynchronous bft made practical. In: Proc. ACM CCS 2018. pp. 2028–2041. ACM (2018)
19. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. JACM **32**(2), 374–382 (1985)
20. Fitzi, M., Garay, J.A.: Efficient player-optimal protocols for strong and differential consensus. In: Proc. ACM PODC 2003. pp. 211–220. ACM (2003)
21. Ganesh, C., Patra, A.: Optimal extension protocols for byzantine broadcast and agreement. Distributed Computing pp. 1–19 (2020)
22. Guo, B., Lu, Z., Tang, Q., Xu, J., Zhang, Z.: Dumbo: Fast asynchronous bft protocols. In: Proc. ACM CCS 2020. ACM (2020)
23. Hendricks, J., Ganger, G.R., Reiter, M.K.: Verifying distributed erasure-coded data. In: Proc. ACM PODC 2007. pp. 139–146. ACM (2007)
24. Kate, A., Goldberg, I.: Distributed key generation for the internet. In: Proc. IEEE ICDCS 2009. pp. 119–128. IEEE (2009)
25. Kursawe, K., Shoup, V.: Optimistic asynchronous atomic broadcast. In: International Colloquium on Automata, Languages, and Programming. pp. 204–215. Springer (2005)
26. Lamport, L.: The weak byzantine generals problem. JACM **30**(3), 668–676 (1983)
27. Lamport, L., Shostak, R., Pease, M.: The byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS) **4**(3), 382–401 (1982)
28. Libert, B., Joye, M., Yung, M.: Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. Theoretical Computer Science **645**, 1–24 (2016)
29. Loss, J., Moran, T.: Combining asynchronous and synchronous byzantine agreement: The best of both worlds. IACR Cryptology ePrint Archive **2018**, 235 (2018)
30. Merkle, R.C.: A digital signature based on a conventional encryption function. In: Eurocrypt 1987. pp. 369–378. Springer (1987)
31. Miller, A., Xia, Y., Croman, K., Shi, E., Song, D.: The honey badger of bft protocols. In: Proc. ACM CCS 2016. pp. 31–42. ACM (2016)
32. Mostéfaoui, A., Moumen, H., Raynal, M.: Signature-free asynchronous byzantine consensus with $t < n/3$ and $o(n^2)$ messages. In: Proc. ACM PODC 2014. pp. 2–9. ACM (2014)

33. Nayak, K., Ren, L., Shi, E., Vaidya, N.H., Xiang, Z.: Improved extension protocols for byzantine broadcast and agreement. In: 34st International Symposium on Distributed Computing (DISC 2020)
34. Neiger, G.: Distributed consensus revisited. Information processing letters **49**(4), 195–201 (1994)
35. Patra, A.: Error-free multi-valued broadcast and byzantine agreement with optimal communication complexity. In: International Conference On Principles of Distributed Systems. pp. 34–49. Springer (2011)
36. Pease, M., Shostak, R., Lamport, L.: Reaching agreement in the presence of faults. JACM **27**(2), 228–234 (1980)
37. Rabin, M.O.: Efficient dispersal of information for security, load balancing, and fault tolerance. JACM **36**(2), 335–348 (1989)
38. Ramasamy, H.V., Cachin, C.: Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In: International Conference On Principles Of Distributed Systems. pp. 88–102. Springer (2005)
39. Shoup, V.: Practical threshold signatures. In: Eurocrypt 2000. pp. 207–220. Springer (2000)
40. Turpin, R., Coan, B.A.: Extending binary byzantine agreement to multivalued byzantine agreement. Information Processing Letters **18**(2), 73–76 (1984)
41. Wensley, J.H., Lamport, L., Goldberg, J., Green, M.W., Levitt, K.N., Melliar-Smith, P.M., Shostak, R.E., Weinstock, C.B.: Sift: Design and analysis of a fault-tolerant computer for aircraft control. Proc. the IEEE **66**(10), 1240–1255 (1978)