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

Yuan Lu
New Jersey Inst. of Tech.
yl768@njit.edu

Zhenliang Lu
New Jersey Inst. of Tech.,
JDD-NJIT-ISCAS Joint Blockchain Lab
zl425@njit.edu

Qiang Tang
New Jersey Inst. of Tech.,
JDD-NJIT-ISCAS Joint Blockchain Lab
qiang@njit.edu

Guiling Wang
New Jersey Inst. of Tech.
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 $O(\ell n^2 + \lambda n^2 + n^3)$ communication (where n is the number of parties, ℓ is the input length, and λ 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 ℓn^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 $O(\ell n + \lambda n^2)$ 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 $O(n^2)$ 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.

CCS CONCEPTS

• **Theory of computation** → **Distributed algorithms**; • **Security and privacy** → *Cryptography*.

ACM Reference Format:
Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. 2020. Dumbo-MVBA: Optimal Multi-Valued Validated Asynchronous Byzantine Agreement, Revisited. In *ACM Symposium on Principles of Distributed Computing (PODC '20), August 3–7, 2020, Virtual Event, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3382734.3405707

PODC '20, August 3–7, 2020, Virtual Event, Italy

ACM ISBN 978-1-4503-7582-5/20/08…$15.00
https://doi.org/10.1145/3382734.3405707

1 INTRODUCTION

Byzantine agreement (BA) was proposed by Lamport, Pease and Shostak in their seminal papers [26, 32] 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 [35]. The original specification of BA [32] 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 [34], which generalizes the particular case of binary BA where input is restricted to either 0 or 1 [26, 30].

Recently, the renewed attention to multi-valued BA is gathered in the *asynchronous* setting [2, 18, 21, 29], 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 [10, 16].

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 [9, 24, 33].

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 $O(1)$ running time and $O(n^2)$ messages, but it incurs $O(\ell n^2 + \lambda n^2 + n^3)$ communicated bits, which is large. Here, n is the number of parties, ℓ represents the bit-length of MVBA input, and λ 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 $O(n)$ 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 $O(n^2)$ 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 ℓ is small (e.g., comparable to λ).

Nevertheless, both of the above MVBA constructions contain the ℓn^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 ℓ is not that small. For instance, Cachin et al. [10] noticed their ABC construction requires the underlying MVBA's input length ℓ to be at least $O(\lambda n)$, as each MVBA input is a set of $(n-f)$ digitally signed ABC inputs. In this case, the ℓn^2 term becomes the dominating factor. For this reason, it was even considered in [18, 29] 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 $\ell \geq O(\lambda n)$.

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

Theorem 1.1. *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 $O(\ell n + \lambda n^2)$ communicated bits and expected constant running time, where ℓ is the input length and λ is a cryptographic security parameter.*

Table 1: Asymptotic performance of MVBA **protocols among n parties with ℓ-bit input and λ-bit security parameter.**

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

† [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 [2, 10] as illustrated in Table 1, but also is *optimal* in the asynchronous setting regarding the following performance metrics[1]:

(1) The execution incurs $O(\ell n + \lambda n^2)$ bits on average, which coincides with the *optimal communication* $O(\ell n)$ when $\ell \geq O(\lambda n)$. This optimality can be seen trivially, since each honest party has to receive the ℓ-bit output, indicating a minimum communication of $\Omega(\ell n)$ bits.
(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 [2, 10], it terminates in expected constant asynchronous rounds with overwhelming probability, which is essentially *asymptotically optimal* for asynchronous BA [3, 19].
(4) As [2, 10], it attains *asymptotically optimal* $O(n^2)$ messages, which meets the lower bound of the messages of optimally-resilient asynchronous BA against adaptive adversary [1, 2].

[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 $O(n^2)$ word communication as [2], because they attain (i) optimal $O(n^2)$ messages and (ii) every message is not larger than a word.

Techniques & challenges. Let us first have a very brief tour of the existing MVBA constructions [2, 10]. 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 $O(n^2)$ 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 $O(n^3)$ communication as the phase that each party receives a $O(n^2)$ size matrix is removed.

We observe that in the first phase of both [2, 10], every party broadcasts its own input to all parties for checking external validity, which already results in ℓn^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.

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, 22] that needs $O(n^2)$ 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 $O(n)$ 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 $O(\ell/n)$ bits, so n dispersals of ℓ-bit input incur only $O(\ell n)$ bits, (ii) the parties can reconstruct a valid value after expected constant number of ABA and recoveries. See detailed discussions in Section 4.

Finally, we present another general self-bootstrap technique Dumbo-MVBA* 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* 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.

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

Strong validity [20, 31] 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 $O(2^\ell)$ asynchronous rounds, where ℓ is the input size in bit.

Weak validity [17, 25], 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, 31], except that they coincide in binary BA [11, 28, 30]. 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 [29].

To circumvent the limits of above validity notions, Cachin et al. [10] proposed *external validity*, which 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 [2]. This succinct construction centers 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 [29]), mainly because the communication cost of the underlying MVBA module contains a quadratic term factored by the MVBA's input length.

[2] There are some other validity notions of multi-valued BA such as vector validity (a.k.a. asynchronous common subset) [5, 16] that are also alternatives to instantiate ABC. We omit discussions about these validity notions, and focus on *external validity*.

2 PROBLEM FORMULATION

2.1 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 [7, 23, 27].

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 λ, which was formulated as *efficiency* in [2, 10] 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 λ.

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.

2.2 Design goal: validated asynchronous BA

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

Definition 2.1. 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 ℓ 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 $O(\lambda n)$.
(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.

2.3 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 $\Omega(n^2)$ [1, 2].
- **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 ℓn term represents a trivial lower bound that all honest parties have to deliver an externally valid value of ℓ bits in length [1, 2, 29], 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 $O(1)$ round complexity is optimal [3, 19].

The above communication, message and round complexities are *probabilistically uniformly bounded* 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.

3 PRELIMINARIES & NOTATIONS

Cryptographic abstractions. Our design uses a few cryptographic primitives/protocols. We briefly describe their high-level abstractions here (see the full version for formal definitions):

- **Erasure code**. A (k, n)-erasure code scheme [6] consists of a tuple of two deterministic algorithms Enc and 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 Dec algorithm.
- **Position-binding vector commitment**. For an established position-binding n-vector commitment (VC), there is a tuple of algorithms (VCom, Open, VerifyOpen). On input a vector $\mathbf{m}$ of any n elements, the algorithm VCom produces a commitment vc for the vector $\mathbf{m}$. On input $\mathbf{m}$ and vc, the Open algorithm can reveal the element m_i committed in vc at the i-th position while producing a short proof π_i, which later can be verified by VerifyOpen. Relying on computational Diffie-Hellman assumption and collision-resistant hash function, there exists a VC scheme [15], such that all above algorithms are deterministic and the length of vc and π_i is $O(\lambda)$-bit.
- **Threshold signature**. 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. In particular, we consider adaptively secure threshold signature scheme, where all partial/full signatures are $O(\lambda)$-bit [27].
- **Common 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. We consider a Coin protocol secure against adaptive adversary, with $O(n^2)$ messages, $O(\lambda n^2)$ bits, and constant running time [28].
- **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**. 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. We consider an ABA protocol secure against adaptive adversary controlling up to $\lfloor \frac{n-1}{3} \rfloor$ parties, with $O(n^2)$ messages, $O(\lambda n^2)$ bits and expected constant running time [28].

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 (MsgType, ID, ...), where ID represents the identifier that tags the protocol instance and MsgType specifies the message type. Moreover, Π[ID] refers to an instance of the protocol Π with identifier ID, and $y \leftarrow \Pi[\text{ID}](x)$ means to invoke Π[ID] on input x and obtain y as output.

4 ASYNCHRONOUS PROVABLE DISPERSAL BROADCAST

The dominating $O(\ell n^2)$ term in the communication complexity of existing MVBA protocols [2, 10] is because every party broadcasts its own input *all* other parties. This turns out to be unnecessary, as in the MVBA protocol, only one single party's input is decided as output. To remedy the needless communication overhead in 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 (APDB), to instantiate the *dispersal-then-recast* idea. The notion is carefully tailored to be efficiently implementable. Especially, in contrast to related AVID protocols [12, 22], APDB can disperse a value at a cost of linear messages instead of $O(n^2)$, as a reflection of following trade-offs:

- The APDB notion weakens AVID, so upon that a party outputs a coded fragment in the dispersal instance of 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 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 $O(\ell/n)$ where ℓ 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 4.1. 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 $\{0,1\}^\ell \cup \perp$.

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 *lock*s, such that the parties can exchange *lock*s 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.

Overview of our 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 π_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 σ_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 σ_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', $store$):
1: parse $store$ as $\langle vc, i, m_i, \pi_i\rangle$
2: return VerifyOpen(vc, m_i, i, π_i) $\wedge$ $i = i'$
function ValidateLock(ID, $lock$):
3: parse $lock$ as $\langle vc, \sigma_1\rangle$
4: return VerifyThld$_{(2f+1)}$($\langle$Stored, ID, $vc\rangle$, σ_1)
function ValidateDone(ID, $done$):
5: parse $done$ as $\langle vc, \sigma_2\rangle$
6: return VerifyThld$_{(2f+1)}$($\langle$Locked, ID, $vc\rangle$, σ_2)

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: $m \leftarrow$ Enc(v), where v is parsed as a $f+1$ vector and m is a n vector
3: $vc \leftarrow$ VCom(m)
4: **for** each $j \in [n]$ **do**
5: $\pi_j \leftarrow$ Open(vc, m_j, j)
6: **let** $store := \langle vc, m_i, i, \pi_i\rangle$ and **send** (Store, ID, $store$) to $\mathcal{P}_j$
7: **wait** until $|S_1| = 2f+1$
8: $\sigma_1 \leftarrow$ Combine$_{(2f+1)}$($\langle$Stored, ID, $vc\rangle$, S_1)
9: **let** $lock := \langle vc, \sigma_1\rangle$ and **multicast** (Lock, ID, $lock$) to all parties
10: **wait** until $|S_2| = 2f+1$
11: $\sigma_2 \leftarrow$ Combine$_{(2f+1)}$($\langle$Locked, ID, $vc\rangle$, S_2)
12: **let** $done := \langle vc, \sigma_2\rangle$ and **deliver** $done$
13: **upon** receiving (Stored, ID, $\rho_{1,j}$) from $\mathcal{P}_j$ for the first time **do**
14: **if** VerifyShare$_{(2f+1)}$($\langle$Stored, 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 (Locked, ID, $\rho_{2,j}$) from $\mathcal{P}_j$ for the first time **do**
17: **if** VerifyShare$_{(2f+1)}$($\langle$Locked, 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 (Store, ID, $store$) from sender $\mathcal{P}_s$ for the first time **do**
20: **if** ValidateStore(i, $store$) = 1 and $stop = 0$ **then**
21: **deliver** $store$ and parse it as $\langle vc, i, m_i, \pi_i\rangle$
22: $\rho_{1,i} \leftarrow$ SignShare$_{(2f+1)}$(sk_i, $\langle$Stored, ID, $vc\rangle$)
23: **send** (Stored, ID, $\rho_{1,i}$) to $\mathcal{P}_s$
24: **upon** receiving (Lock, ID, $lock$) from sender $\mathcal{P}_s$ for the first time **do**
25: **if** ValidateLock(ID, $lock$) = 1 and $stop = 0$ **then**
26: **deliver** $lock$ and parse it as $\langle vc, \sigma_1\rangle$
27: $\rho_{2,i} \leftarrow$ SignShare$_{(2f+1)}$(sk_i, $\langle$Locked, ID, $vc\rangle$)
28: **send** (Locked, ID, $\rho_{2,i}$) to $\mathcal{P}_s$
procedure $abandon$(ID):
29: $stop \leftarrow 1$

Details of our 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).

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

let $C \leftarrow [\,]$
1: **upon** receiving input ($store$, $lock$) **do**
2: **if** $lock \neq \emptyset$ **then**
3: **multicast** (RcLock, ID, $lock$) to all
4: **if** $store \neq \emptyset$ **then**
5: **multicast** (RcStore, ID, $store$) to all
6: **upon** receiving (RcLock, ID, $lock$) **do**
7: **if** ValidateLock(ID, $lock$) = 1 **then**
8: **multicast** (RcLock, ID, $lock$) to all, if was not sent before
9: parse $lock$ as $\langle vc, \sigma_1\rangle$
10: **wait** until $|C[vc]| = f+1$
11: $v \leftarrow$ Dec($C[vc]$)
12: **if** VCom(Enc(v)) = vc **then return** v
13: **else return** $\perp$
14: **upon** receiving (RcStore, ID, $store$) from $\mathcal{P}_j$ for the first time **do**
15: **if** ValidateStore(j, $store$) = 1 **then**
16: parse $store$ as $\langle vc, m_j, j, \pi_j\rangle$
17: $C[vc] \leftarrow C[vc] \cup (j, m_j)$
18: **else discard** the invalid message

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 π_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 $\langle$Stored, 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 σ_1 for the string $\langle$Stored, ID, $vc\rangle$. Then, $\mathcal{P}_s$ sends *lock* including vc and σ_1 to all parties through Lock messages. Upon receiving Lock message, $\mathcal{P}_i$ verifies whether σ_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 $\langle$Locked, 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 σ_2 for $\langle$Locked, 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.

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 $\perp$).

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 detailed proofs are deferred to the full version.

Complexities. The complexities of our APDB construction can be briefly analyzed as: (i) The PD subprotocol has 4 one-to-all (or all-to-one) rounds, which attains $O(n)$ messages and $O(1)$ running time; in addition, each message in PD contains at most $O(\ell/n + \lambda)$ bits, including: a fragment of input having $O(\ell/n)$ bits, a vector commitment having $O(\lambda)$ bits, and an openness proof having $O(\lambda)$ bits, so the overall bits of the $O(n)$ messages in PD are $O(\ell + \lambda n)$; (ii) The RC subprotocol only has one all-to-all round, which incurs $O(n^2)$ messages and constant running time; moreover, each message in RC contains at most $O(\ell/n+\lambda)$ bits, as it has at most a $O(\ell/n)$-bit fragment, a $O(\lambda)$-bit commitment, and a $O(\lambda)$-bit openness proof, such that the total communicated bits in each RC are $O(n\ell + n^2\lambda)$.

5 AN OPTIMAL MVBA PROTOCOL FROM ABA

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 $O(\ell n + \lambda n^2)$ bits, which is asymptotically better than all previous results [2, 10] and optimal for sufficiently large input.

5.1 Overview of Dumbo-MVBA

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.

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 PD[⟨id, j⟩]

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

13: **upon** PD[⟨id, j⟩] delivers $store$ **do**
14: $store[j] \leftarrow store$

15: **upon** PD[⟨id, j⟩] delivers $lock$ **do**
16: $lock[j] \leftarrow lock$

17: **upon** PD[⟨id, i⟩] delivers $done$ **do**
18: **multicast** (DONE, id, $done$)

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

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

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

36: **upon** receiving (RCBALLOTPREPARE, id, l, $lock$) from $\mathcal{P}_j$ **do**
37: **if** ValidateLock(⟨id, l⟩, $lock$) = 1 **then**
38: $lock[l] \leftarrow lock$
39: $rc\text{-}ballot[l] \leftarrow 1$

5.2 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 $[n]$.

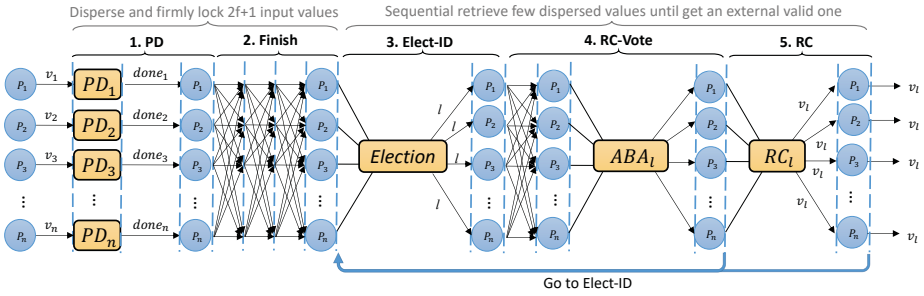

Figure 1: The execution flow of Dumbo-MVBA.

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, (PD[ID], RC[ID] represents a pair of (PD, RC) instance with identifier ID, where ID := $\langle \text{id}, i \rangle$ 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 ID := $\langle \text{id}, k \rangle$ 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 PD[$\langle \text{id}, i \rangle$], 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 (PD[$\langle \text{id}, l \rangle$], RC[$\langle \text{id}, l \rangle$]) 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 RC[$\langle \text{id}, l \rangle$] instance or not. This phase has to cope with a major limit of RC subprotocol, that the RC[$\langle \text{id}, l \rangle$] 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 PD[$\langle \text{id}, l \rangle$] 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 ABA[$\langle \text{id}, l \rangle$] with input 1, otherwise, it invokes ABA[$\langle \text{id}, l \rangle$] with input 0. The above design follows the idea of biased validated binary agreement presented by Cachin et al. in [10], and ABA[$\langle \text{id}, l \rangle$] 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 ABA[$\langle \text{id}, l \rangle$] returns 1, all honest parties would enter this phase and there is always at least one honest party has delivered the valid *lock* regarding RC[$\langle \text{id}, l \rangle$]. As such, the parties can always invoke the corresponding RC[$\langle \text{id}, l \rangle$] 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.

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 defer the detailed proofs to the full paper and 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 dispersals, which prevents the adaptive adversary from "tampering" the values dispersed by uncorrupted parties. Moreover, 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 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.

Complexities. In addition, Dumbo-MVBA achieves: (i) asymptotically optimal round and message complexities, and (ii) asymptotically optimal communicated bits $O(\ell n + \lambda n^2)$ for any input $\ell \geq \lambda n$. The breakdown of its cost can be briefly summarized as:

- The dispersal phase (i.e., n PD instances) terminates in constant time, with $O(n^2)$ messages and $O(\ell n + \lambda n^2)$ bits, since each PD instance terminates in constant time with $O(n)$ messages and $O(\ell + \lambda n)$ bits; the finish phase consists of 3 all-to-all broadcasts, which terminates in constant time with $O(\lambda n^2)$ bits and $O(n^2)$ messages.
- With the sequential repetition of the elect-ID phase, the recast-vote phase and the recast phase, the probability of not terminating decreases exponentially. In particular, the elect-ID phase (i.e., a common coin) terminates in constant time with $O(n^2)$ messages and $O(\lambda n^2)$ bits, the recast-vote phase (i.e., an all-to-all broadcast plus an ABA instance) returns in expected constant time with $O(n^2)$ messages and $O(\lambda n^2)$ bits, and the recast phase (i.e., a RC instance) halts in constant time with $O(n^2)$ messages and $O(\ell n + \lambda n^2)$ bits. Moreover, the elect-ID and recast-vote phases are repeated for three times on average, while the recast-phase is invoked twice on average. To sum up, the sequential repetition of these phases would incur only expected constant running time, $O(n^2)$ messages, and $O(\ell n + \lambda n^2)$ bits.

6 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).

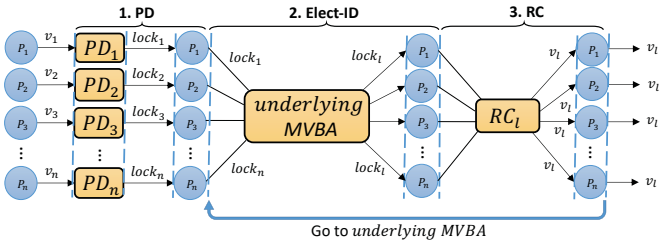

Figure 2: The execution flow of Dumbo-MVBA*.

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 $O(n\ell + n^2\lambda)$ communicated bits, without giving up the optimal running time and message complexity.

Protocol execution. Here is our generic Dumbo-MVBA* framework. Informally, a Dumbo-MVBA* instance with identification id (see the formal description in the full paper) proceeds as:

(1) *Dispersal phase* (line 1-3, 8-11). n concurrent PD instances are activated. Each party $\mathcal{P}_i$ is the designated sender of the instance PD[$\langle$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 PD[$\langle$id, $i\rangle$], it takes the proof $lock_i$ as input to invoke a concrete MVBA instance with identifier $\langle$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 PD[$\langle$id, $l\rangle$].
(3) *Recast phase* (line 6-7). Eventually, the MVBA[$\langle$id, $k\rangle$] instance returns to all parties a common $lock_l$ proof for the PD[$\langle$id, $l\rangle$] instance, namely, MVBA elects a party $\mathcal{P}_l$ to recover its dispersal. Then, all honest parties invoke RC[$\langle$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 MVBA[$\langle$id, $k+1\rangle$] instance with still passing $lock_i$ as input, until a valid v_l can be recovered by an elected RC[$\langle$id, $l\rangle$] instance.

Note 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.

Security & complexities. Dumbo-MVBA* realizes (optimal) MVBA among n parties against adaptive adversary controlling $f \le \lfloor \frac{n-1}{3} \rfloor$ 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. We defer the detailed proofs for the hereinabove conclusions to the full version.

The cost of Dumbo-MVBA* is incurred by: (i) n concurrent PD instances; (ii) few expected constant number (i.e., two) of underlying MVBA instances; (iii) few expected constant number (i.e., two) of RC instances. Recall the complexities of PD and RC protocols: PD costs $O(n)$ messages, $O(\ell + n\lambda)$ bits, and $O(1)$ running time; RC costs $O(n^2)$ messages, $O(n\ell + n^2\lambda)$ bits, and $O(1)$ running time. Suppose the underlying MVBA module incurs expected $O(\text{poly}_{rt}(n))$ running time, expected $O(\text{poly}_{mc}(n))$ messages, and expected $O(\text{poly}_{cc}(\ell, \lambda, n))$ bits, where $O(\text{poly}_{mc}(n)) \ge O(n^2)$ and $O(\text{poly}_{cc}(\ell, \lambda, n)) \ge O(\ell n + n^2)$ due to the lower bounds of adaptively secure MVBA. Thus the complexities of Dumbo-MVBA* are:

- **Running time:** Since PD and RC are deterministic protocols with constant running timing, the running time of Dumbo-MVBA* is dominated by the underlying MVBA module, namely, $O(\text{poly}_{rt}(n))$.

- **Message complexity:** The message complexity of n PD instances (or a RC instance) is $O(n^2)$. The message complexity of the underlying MVBA is $O(\text{poly}_{mc}(n))$, where $O(\text{poly}_{mc}(n)) \geq O(n^2)$. As such, the messages complexity of Dumbo-MVBA* is dominated by the underlying MVBA protocol, namely, $O(\text{poly}_{mc}(n))$.
- **Communication complexity:** The communication of n concurrent PD instances (or a RC instance) is $O(n\ell + n^2\lambda)$. The underlying MVBA module incurs $O(\text{poly}_{cc}(\lambda, \lambda, n))$ bits. So the overall communication complexity of Dumbo-MVBA* is $O(\ell n + \lambda n^2 + \text{poly}_{cc}(\lambda, \lambda, n))$.

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

Concrete instantiation. Dumbo-MVBA* can be instantiated by extending the MVBA protocol of Abraham et al. [2]. Moreover, Abraham et al. achieved expected $O(1)$ running time, $O(n^2)$ messages and $O(n^2\ell + n^2\lambda)$ bits, and our Dumbo-MVBA* framework can extend their result to attain $O(n\ell + n^2\lambda)$ bits without scarifying the optimal running time and message complexity, which therefore instantiates *optimal* MVBA for sufficient input length $\ell \geq O(n\lambda)$.

7 CONCLUSION

We present two MVBA protocols that reduce the communication cost of prior art [2, 10] by an $O(n)$ 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 [29]. Moreover, they can provide better building blocks for the Dumbo BFT protocols [21], 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.

ACKNOWLEDGMENTS

Qiang and Zhenliang are supported in part by JD Digits. We would like to thank the anonymous reviewers for their valuable comments and suggestions about this paper.

REFERENCES

[1] Ittai Abraham, TH Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. 2019. Communication complexity of byzantine agreement, revisited. In *Proc. ACM PODC 2019*. 317–326.
[2] Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. 2019. Asymptotically optimal validated asynchronous byzantine agreement. In *Proc. ACM PODC 2019*. 337–346.
[3] Michael Ben-Or. 1983. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proc. ACM PODC 1983*. 27–30.
[4] Michael Ben-Or and Ran El-Yaniv. 2003. Resilient-optimal interactive consistency in constant time. *Distributed Computing* 16, 4 (2003), 249–262.
[5] Michael Ben-Or, Boaz Kelmer, and Tal Rabin. 1994. Asynchronous secure computations with optimal resilience. In *Proc. ACM PODC 1994*. 183–192.
[6] Richard E Blahut. 1983. *Theory and practice of error control codes*. Addison-Wesley.
[7] Alexandra Boldyreva. 2003. Threshold signatures, multisignatures and blind signatures based on the gap-Diffie-Hellman-group signature scheme. In *PKC 2003*. Springer, 31–46.
[8] Gabriel Bracha. 1987. Asynchronous Byzantine agreement protocols. *Information and Computation* 75, 2 (1987), 130–143.
[9] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. 2002. Asynchronous verifiable secret sharing and proactive cryptosystems. In *Proc. ACM CCS 2002*. 88–97.
[10] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and efficient asynchronous broadcast protocols. In *Annual International Cryptology Conference*. Springer, 524–541.
[11] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2005. Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography. *Journal of Cryptology* 18, 3 (2005), 219–246.
[12] Christian Cachin and Stefano Tessaro. [n.d.]. Asynchronous verifiable information dispersal. In *Proc. IEEE SRDS 2005*. 191–201.
[13] Christian Cachin and Marko Vukolic. 2017. Blockchain Consensus Protocols in the Wild (Keynote Talk). In *31st International Symposium on Distributed Computing (DISC 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[14] Ran Canetti and Tal Rabin. 1993. Fast asynchronous Byzantine agreement with optimal resilience. In *Proc. ACM STOC 1993*. 42–51.
[15] Dario Catalano and Dario Fiore. 2013. Vector commitments and their applications. In *PKC 2013*. Springer, 55–72.
[16] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2006. From consensus to atomic broadcast: Time-free Byzantine-resistant protocols without signatures. *Comput. J.* 49, 1 (2006), 82–96.
[17] Danny Dolev and H. Raymond Strong. 1983. Authenticated algorithms for Byzantine agreement. *SIAM J. Comput.* 12, 4 (1983), 656–666.
[18] Sisi Duan, Michael K Reiter, and Haibin Zhang. 2018. BEAT: Asynchronous BFT made practical. In *Proc. ACM CCS 2018*. 2028–2041.
[19] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *JACM* 32, 2 (1985), 374–382.
[20] Matthias Fitzi and Juan A Garay. 2003. Efficient player-optimal protocols for strong and differential consensus. In *Proc. ACM PODC 2003*. 211–220.
[21] Binyong Guo, Zhenliang Lu, Qiang Tang, Jing Xu, and Zhenfeng Zhang. 2020. Dumbo: Fast Asynchronous BFT Protocols. In *Proc. ACM CCS 2020*.
[22] James Hendricks, Gregory R Ganger, and Michael K Reiter. 2007. Verifying distributed erasure-coded data. In *Proc. ACM PODC 2007*. 139–146.
[23] Aniket Kate and Ian Goldberg. 2009. Distributed key generation for the internet. In *Proc. IEEE ICDCS 2009*. 119–128.
[24] Klaus Kursawe and Victor Shoup. 2005. Optimistic asynchronous atomic broadcast. In *International Colloquium on Automata, Languages, and Programming*. 204–215.
[25] Leslie Lamport. 1983. The weak Byzantine generals problem. *JACM* 30, 3 (1983), 668–676.
[26] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4, 3 (1982), 382–401.
[27] Benoît Libert, Marc Joye, and Moti Yung. 2016. Born and raised distributively: Fully distributed non-interactive adaptively-secure threshold signatures with short shares. *Theoretical Computer Science* 645 (2016), 1–24.
[28] Julian Loss and Tal Moran. 2018. Combining Asynchronous and Synchronous Byzantine Agreement: The Best of Both Worlds. *IACR Cryptology ePrint Archive* 2018 (2018), 235.
[29] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. 2016. The honey badger of BFT protocols. In *Proc. ACM CCS 2016*. 31–42.
[30] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. 2014. Signature-free asynchronous byzantine consensus with $t < n/3$ and $O(n^2)$ messages. In *Proc. ACM PODC 2014*. 2–9.
[31] Gil Neiger. 1994. Distributed consensus revisited. *Information processing letters* 49, 4 (1994), 195–201.
[32] Marshall Pease, Robert Shostak, and Leslie Lamport. 1980. Reaching agreement in the presence of faults. *JACM* 27, 2 (1980), 228–234.
[33] HariGovind V Ramasamy and Christian Cachin. 2005. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *International Conference On Principles Of Distributed Systems*. 88–102.
[34] Russell Turpin and Brian A Coan. 1984. Extending binary Byzantine agreement to multivalued Byzantine agreement. *Inform. Process. Lett.* 18, 2 (1984), 73–76.
[35] John H Wensley, Leslie Lamport, Jack Goldberg, Milton W Green, Karl N Levitt, Po Mo Melliar-Smith, Robert E Shostak, and Charles B Weinstock. 1978. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proc. the IEEE* 66, 10 (1978), 1240–1255.