DARE to agree: Byzantine Agreement with Optimal Resilience and Adaptive Communication [*]

PIERRE CIVIT, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
MUHAMMAD AYAZ DZULFIKAR, NUS Singapore, Singapore
SETH GILBERT, NUS Singapore, Singapore
RACHID GUERRAOUI, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
JOVAN KOMATOVIC, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
MANUEL VIDIGUEIRA, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Byzantine Agreement (BA) enables n processes to reach consensus on a common valid L_o-bit value, even in the presence of up to $t < n$ faulty processes that can deviate arbitrarily from their prescribed protocol. Despite its significance, the optimal communication complexity for key variations of BA has not been determined within the honest majority regime ($n = 2t + 1$), for both the worst-case scenario and the adaptive scenario, which accounts for the actual number $f \leq t$ of failures. We introduce ADA-DARE (Adaptively Disperse, Agree, Retrieve), a novel universal approach to solve BA efficiently. Let κ represent the size of the cryptographic objects required to solve BA when $t > n/3$. Different instantiations of ADA-DARE achieve near-optimal adaptive bit complexity of $O(nL_o + n(f + 1)\kappa)$ for both strong multi-valued validated BA (SMVBA) and interactive consistency (IC). By definition, for IC, $L_o = nL_{in}$, with L_{in} representing the size of an input value. These results achieve optimal $O(n(L_o + f))$ word complexity and significantly improve the previous best results by up to a linear factor, depending on L_o and f.

1 INTRODUCTION

Byzantine Agreement (BA) is a core primitive of distributed computing [64]. It is indispensable to state machine replication (SMR) [1, 18, 54, 60], blockchain systems [3, 15, 33, 34, 44], and various other distributed protocols [7, 40, 45, 47]. In BA, n processes propose an L_{in}-bit value and agree on an L_o-bit value, while tolerating up to t arbitrary failures. If a process exhibits an arbitrary failure, the process is said to be faulty; otherwise, the process is said to be correct. Formally, a Byzantine Agreement protocol satisfies the following guarantees:

- *Termination:* All correct processes eventually decide.
- *Agreement:* No two correct processes decide different values.

To avoid trivial agreement on a pre-established value, BA necessitates an additional *Validity* property, delineating permissible decisions.

- *Interactive consistency* (IC) requires that the output corresponds to a vector containing the proposals of all (honest) processes.
- *Strong Validity* (SBA) dictates that, if all correct processes propose the same value, that value must be decided.
- Validated BA (VBA) ensures *External Validity*, stipulating that a decided value must satisfy a globally verifiable predicate.
- *Weak Validity* (WBA) ensures that if all processes are correct and unanimously propose the same value, that value is the unique admissible decision.

[*]This paper represents an extension of [27]. There are three crucial differences between [27] and this paper: (1) [27] solves multi-valued Byzantine agreement with only strong validity (i.e., it does not tackle interactive consistency), (2) [27] focuses only on constant-sized value and achieves $O\big(n(f + 1)\big)$ word complexity (or, equivalently, $O\big(n(f + 1)\kappa\big)$ bit complexity), and (3) [27] achieves only sub-optimal resilience ($n = (2 + \epsilon)t + 1$, where t denotes the number of tolerated failures).

External validity is often implicitly associated with Weak Validity (WBA).

In general, the running time and the communication complexity of Byzantine Agreement depend on the number of nodes n, the size of the values being agreed upon L_o, and the number of failures that the protocol can tolerate t. In some executions, the real number of failures f may be significantly less than the maximum t that the protocol can tolerate. In such cases, we can design protocols that are significantly more efficient [71]. In this paper, we are primarily interested in *adaptive* protocols where the complexity depends on $f \leq t$, the actual number of failures in an execution.

What is the best communication complexity that can be achieved? Negative results, à la Dolev and Reischuk, essentially show that a quadratic number of bits must be exchanged [24, 37, 71] to deterministically solve BA. In fact, solving deterministic Strong Byzantine Agreement (SBA) requires at the very least a resiliency threshold of $t < n/2$, and the exchange of $\Omega(nL_o + nf)$ bits [71], where $f \leq t$ is the actual number of failures that occur in the execution.

Positive results have demonstrated BA protocols that achieve (near-)optimality for some specific validity properties and valuations of the parameters f, t, L_o, L_{in} [5, 21, 32, 38, 61, 62]. Nonetheless, it has remained an open question whether there exists a protocol with optimal resilience and optimal communication complexity that adapts to the actual number of failures.

Firstly, we are unaware of any adaptive *Strong* BA protocols with optimal $n/2$ resilience. In a recent exciting result, Elsheimy, Tsimos, and Papamanthou developed the first adaptive SBA protocol—with resilience $t < (1/2 - \epsilon)n$. Their result relies on a clever construction of reliable voting and Weak Byzantine Agreement, and uses a strong crypto primitive: the SNARK. Our goal in this paper is to provide a protocol with optimal resilience $t < n/2$ (ideally without using SNARKs).

Second, existing adaptive BA algorithms are not efficient with respect to the size of the agreed upon value L_o. Elsheimy, Tsimos, and Papamanthou [38] provide an adaptive Strong BA protocol with word complexity of $O(nL_o(f+1))$; Cohen, Keidar, and Spiegelman [32] provide an adaptive Weak BA protocol with word complexity $O(nL_o(f+1))$. Our goal in this paper is to provide a protocol with word complexity $O(nL_o + nf)$.

Third, the most efficient interactive consistency protocol to date is a recent paper [5] by Abraham, Nayak, and Shrestha, which achieves $O(nL_o + \kappa n^3)$ communication (using an interesting new version of gradecast). Our goal here is to provide an *adaptive* protocol with word complexity $O(nL_o + nf)$.

Protocol	Problem	Resiliency	Words	Crypto
[32]([61])	W+E	$n/2$	$O(nL_o(f+1))$	TSS
This - 1	W+E	$n/2$	$O(nL_o + nf)$	TSS
[38]	S	$n/(2+\epsilon)$	$O(nL_o(f+1))$	TSS, SNARK
[62]([61])+[5]	S+E	$n/2$	$O(nL_o + nt)$	TSS
This - 2	S+E	$n/2$	$O(nL_o + nf)$	MTSS†
[5]	IC	$n/2$	$O(nL_o + n^3)$	TSS
This - 3	IC	$n/2$	$O(nL_o + nf)$	TSS

Fig. 1. Performance of various consensus algorithms with L_o-bit values and κ-bit size of cryptographic objects. $\epsilon \in \Omega(1)$ denotes an arbitrarily small constant. W, S, and E refer to weak, strong and external-validity respectively, where $L_o = L_i$. IC refers to interactive consistency, where $L_o = nL_i$ by definition. We assume that $\kappa > \log(n)$ to avoid an adversary with an exponential computational power, and collision between cryptographic signatures. We define a *word* to be a string of $O(\kappa)$ bits. †Multiverse Threshold Signature Scheme (MTSS) [55] is a cryptographic scheme strictly lighter than a generic SNARK.

Our approach. We apply the DARE paradigm (Disperse-Agree-REtrieve) [23] to develop Byzantine Agreement protocols with adaptive communication complexity, i.e., whose communication cost depends on the number of faulty processes $f \leq t$. The basic idea is to first disperse the value to be agreed upon, deriving a digest of the value and a proof-of-dispersal; this guarantees that the value is valid and that it is available to at least $t + 1 - f$ processes. We then use an existing consensus protocol to agree on a digest. Finally, each process retrieves the value associated with the agreed-upon digest.

There are three main technical innovations involved in our results, namely (1) verifiable vector collection, (2) persuasion via bucket certificates, and (3) adaptive retrieval.

First, we can only disperse valid values, and we can only determine if a value is valid if we have enough information on the other inputs to the system. At the same time, it is too expensive for all n processes to collect all n inputs of size L_{in}. Hence, we design a Verifiable Vector Collection protocol that ensures that honest leaders will efficiently collect enough verifiable information to determine validity; our protocol is based on the idea of "LearnOrExpose": in every view with an honest leader, either more (missing) information is collectively learned, or a malicious process is exposed.[1]

Second, we design a *persuasion* mechanism for proving that a digest is associated with a valid value. This mechanism is based on a new type of "Bucket Certificate"—inspired by the idea of [67] (which was designed for $n/3$ resilience). A bucket certificate can efficiently guarantee that a value satisfies strong validity. The key challenge is to succinctly and efficiently prove a negative: that not all honest processes input the same value even when we do not know which processes are honest.

Third, we develop a new adaptive retrieval protocol, based on fault-tolerant expander graphs and Reed-Solomon codes, to ensure that all correct processes can retrieve the final agreed-upon value. The problem is that only some honest processes ($t + 1 - f$) know the required value, and yet it is too expensive for them to broadcast it directly to everyone. Thus a careful combination of active dissemination (via an expander) and carefully requested Reed-Solomon shares ensures the proper adaptive complexity.

Our results. Putting these pieces together yields the following, which are summarized in Table 1.

(1) For weak validity and external validity, we provide optimal word complexity: $O(nL_o + nf)$.
(2) For strong validity (and external validity), we provide optimal word complexity with optimal resilience ($t < n/2$), without using SNARKs: $O(nL_o + nf)$.
(3) For interactive consistency, we provide optimal word complexity: $O(nL_o + nf)$.

(Note that a word is of size $O(\kappa)$, where κ is the size of the cryptographic elements.)

Related Work

We now discuss related work.

Deterministic Byzantine agreement. In their foundational work, Dolev and Reischuk [37] demonstrated quadratic lower bounds on the communication complexity for deterministic Byzantine broadcast (and by extension, strong consensus). In the authenticated setting (utilizing ideal digital signatures [17]), Byzantine broadcast must exchange at least $\Omega(nt)$ signatures and $\Omega(n + t^2)$ messages in some failure-free execution ($f = 0$). Spiegelman [71] later extended this lower bound to cover adaptive scenarios, showing $\Omega(n + t(f + 1))$ words are necessarily exchanged, regardless of the employed cryptographic schemes.

This quadratic communication complexity has been proven optimal for $f = \Theta(t)$ across various contexts [5, 8, 21, 28, 61, 62]. In some cases, using threshold signatures [70], which extend beyond

[1]This takes some inspiration from previous ideas of accountability [48].

the idealized authenticated model. Additionally, an amortized communication cost of $O(n)$ messages has been achieved in multi-shot Byzantine broadcast [74].

However, the possibility of reducing word complexity to $o(n^3)$ for interactive consistency remains an open question (1) under a dishonest majority and (2) against an unbounded adversary (assuming $t < n/3$ [39]). Deterministic quadratic BA protocols have also been developed for partially synchronous environments [22, 23, 25, 56, 57, 75].

Circumventing the quadratic lower bound with randomization. Even with randomization, no Byzantine broadcast algorithm can achieve sub-quadratic expected message complexity against a strongly rushing adaptive adversary equipped with after-the-fact message removal capabilities [2]. Yet, it is feasible to construct randomized Byzantine agreement algorithms that achieve sub-quadratic expected communication complexity when facing a weaker adversary [2, 9, 13, 13, 14, 20, 42, 43, 51–53, 66]. This also extends to partial synchrony [2, 9, 20, 66, 69] and even to complete asynchrony, assuming either a private setup [11] or a delayed adversary [31]. Moreover, fully-asynchronous quadratic BA is achievable with fewer assumptions [4, 59, 62].

In the dishonest majority regime ($t > n/2$), Blum et al. [10] delineated new lower bounds, showing that a randomized Byzantine broadcast protocols with merely $O(1)$ correct processes, necessarily exchange $\Omega(n^2)$ bits in expectation. Recent advancements approach this bound [19, 72].

Circumventing the quadratic lower bound with adaptiveness. Even with advanced cryptography, an $\Omega(n^2 L_{in})$ price must be paid in failure-free execution to achieve interactive consistency, since every correct process must receive the input of every correct process.

However, using sophisticated cryptographic schemes, like threshold signatures [70], allows for *adaptive* BA protocols. These protocols ensure a cheaper validity property, including external validity [32, 71], binary strong validity [38], and Byzantine broadcast [32]. Typically, these approaches leverage the algorithm from [61] as a fallback for pessimistic scenarios.

Organization of the paper. We provide a technical overview in §2. We state the full formal system model and preliminaries in §3. In §4, we present the main building blocks of our DARE-style composition. §5 and §6 respectively detail our efficient retrieval and dispersal protocols. The optional appendix contains omitted proofs.

2 TECHNICAL OVERVIEW

In this section, we give an overview of the technical details of ADA-DARE.

Silent views. Most of our sub-protocols, along with existing adaptive BA protocols, adhere to the *silent views* framework introduced by Spiegelman [71]. This framework takes the classical view-based approach, while attempting to induce silent views to save message complexity once agreement has been achieved. These protocol operate across up to n views, with each view led by a round-robin leader. The communication follows a "leader-to-all, all-to-leader" pattern. In each view, the leader's objective is to drive agreement on an admissible output. Within $O(f)$ views, a correct leader ensures that all honest parties have output an admissible value. Subsequent honest leaders refrain from communicating in their views, thereby remaining 'silent', leading to a word complexity of $O(n \cdot f)$. Recently, Cohen, Keidar, and Spiegelman [32] have used this paradigm to develop a VBA protocol, called **CKS** in this paper[2], achieving adaptive $O(nL_o(f+1))$ word complexity.

Disperse/Agree/Retrieve. ADA-DARE efficiently handles long inputs by adapting the classic Disperse/Agree/Retrieve paradigm [30] (see Figure 2). In this approach: (1) a long input v is

[2]The version of **CKS** used in this paper incorporates the correction of a technical issue proposed by [38], and minor modifications, discussed herein, to ensure External Validity.

represented as a polynomial $P_v(\cdot)$ of degree t within a Galois field, where the polynomial's evaluations $\{P_v(i)\}_{i\in[1:n]}$ (i.e., symbols or shares) are dispersed. A correct dispersion is attested by a *proof-of-dispersal* (PoD), guaranteeing that the corresponding value is both (a) valid, and (b) efficiently retrievable. In ADA-DARE, efficient retrievability is ensured if the corresponding value has been observed by at least $t+1-f$ correct processes. (2) Processes reach consensus on a cryptographic digest $d = \text{Digest}(v)$ of the lengthy input, in which d must be backed by a corresponding PoD. (3) The original input v is recovered through the exchange of consistent polynomial shares and subsequent polynomial interpolation.

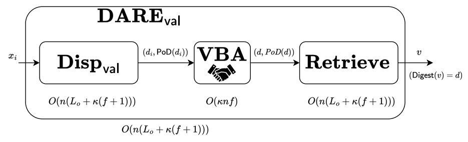


Fig. 2. ADA-DARE for any validity property val with the DAR paradigm.

Adaptive Agreement & Retrieval. Our first challenge is to ensure adaptive communication complexity in agreement and retrieval. Suppose that each correct process starts with some digest d_i, backed by a corresponding PoD ($O(\kappa)$-bits in total).

In the agreement phase, consensus is executed on a digest d linked to its PoD. This is achieved with $n/2$ resiliency and adaptive $O(n(f+1)\kappa)$ communication complexity via the **CKS** protocol.

Next, retrieval consists of two phases (see Figure 3). In the *to almost everywhere* phase, correct processes that have obtained the value v corresponding to digest d gossip using an *expander graph*. Specifically, we use the expander construction from [73] with some nice properties: it has a constant degree and, after enough time (proportional to the diameter of the graph), all but $O(f)$ correct processes will have obtained v. Next, in the *to everywhere* phase, each correct process p_i that has not yet obtained the value (v) tries to obtain it in two steps. First, p_i asks for its own share (i.e., the i-th share, $P_v(i)$) from all other processes. Then, p_i asks for the j-th share from process p_j. Importantly, each share is accompanied by an accumulator witness, allowing a correct process to verify the correctness of the share. By the end of the first step, every correct process p_j will know their respective share. Therefore, at the end of the second phase, p_i will be able to collect $t+1$ different shares, allowing it to recover v. In total, the retrieval phase exchanges $O(nL_o + n(f+1)\kappa)$ bits.

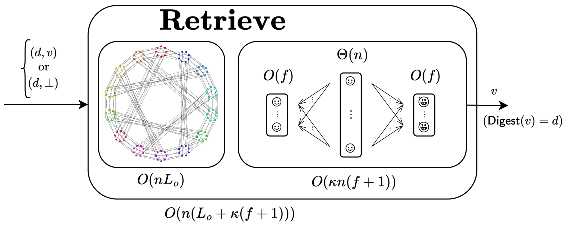
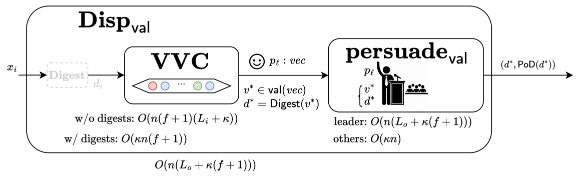


Fig. 3. Retrieve. In the first phase, processes gossip over an expander graph; if sufficient processes obtained v, then all but $O(f)$ correct process will obtain v. In the second phase, each process that has not obtained v attempts to recover v by (1) obtaining its own share of v and then (2) obtaining $t+1$ different shares.

Verifiable vector collection & persuasion: key building block of efficient Dispersal. Let us start by explaining why obtaining proof that a value is admissible can be difficult, using strong validity as an example. Suppose that a correct process manages to collect proposals from $n = 2t + 1$ processes. In this case, the collected proposals themselves easily serve as proof: if a majority value v exists then the proposals serve as v's proof, and if not, they also serve as proof that any value is valid. However, suppose that even one proposal is missing. As it turns out, it might be that nothing can be proven by these $n - 1 = 2t$ proposals. For example, suppose the $2t$ proposals consist of t proposals for value 0 and t proposals for value 1. We can then not distinguish the cases where 1) the missing proposal corresponds to a faulty process, 2) the missing proposal corresponds to a correct process that proposes 0 (and all correct processes propose 0), and 3) the missing proposal corresponds to a correct process that proposes 1 (and all correct processes propose 1). In some sense, missing even just one proposal makes it difficult to show that a value is indeed admissible. This is the main motivation behind *verifiable vector collection* (VVC), which enables a correct process to collect a vector of n "input materials". Here, an input material can either be a signed message, containing an *entry* (a value or its digest, as desired, of L_e-bits) from the corresponding process, or a *proof-of-misbehavior* (PoM), which proves that the corresponding process is faulty. Importantly, a PoM can only be formed for a faulty process: a correct process can never be accused. Then, suppose a correct process has succeeded in VVC. The next step would be to efficiently persuade other processes that a certain value v is indeed admissible, and obtain a PoD for its digest. This is done by a subprotocol named persuade, which is tailored individually to specific validity properties (such as strong validity and interactive consistency), whose goal is to efficiently form a PoD through the collected verifiable vector. Based on these ideas, we devise ADA-DISP, a dispersal protocol that works in a leader-based paradigm. Within each view, a leader attempts to collect a verifiable vector and build a PoD. The protocol is illustrated in Figure 4.



$\mathbf{Disp}_{\mathsf{val}}$
x_i
Digest
d_i
VVC
$p_\ell : vec$
$v^* \in \mathsf{val}(vec)$
$d^* = \mathsf{Digest}(v^*)$
$\mathbf{persuade}_{\mathsf{val}}$
p_ℓ
v^*
d^*
$(d^*, \mathsf{PoD}(d^*))$
w/o digests: $O(n(f+1)(L_i+\kappa))$
w/ digests: $O(\kappa n(f+1))$
leader: $O(n(L_o+\kappa(f+1)))$
others: $O(\kappa n)$
$O(n(L_o+\kappa(f+1)))$

Fig. 4. Disp_{val} for any validity property val. The protocol is built on top of an efficient verifiable vector collection (VVC) and an efficient persuasion sub-protocol.

Adaptive verifiable vector collection (VVC). Recall that VVC is contained in our dispersal protocol. Intuitively, each process sends its signed entry to the leader in every view. To tackle the problem of uncooperative faulty processes, we introduce the LearnOrExpose primitive. In this primitive all processes engage with an (allegedly faulty) process accused by a leader. Under a correct leader, an accused process either (i) (Learn) has its signed entry forwarded to the leader, or (ii) (Expose) is confronted with an undeniable PoM via a $(t+1)$-threshold signature. Finally, the leader broadcasts the indirect answer (or the PoM). After each unsuccessful view driven by a correct leader, the number of accused Byzantine processes, linked with corresponding indirect answers (or PoM), is incremented by one. Hence, a complete collection, and so a successful view, is reached by view $2f+1$, after at most f Byzantine leaders, and f unsuccessful correct leaders.[3] Through careful bookkeeping of previous interactions, the primitive collectively exchanges $O(nf(L_e+\kappa))$ bits. Therefore, over $2f+1$ views, VVC exchanges $O(n(f+1)(L_e+\kappa)) \leq O(nL_o + n(f+1)\kappa)$ bits.

Efficient persuasion for interactive consistency (IC). Suppose we have reached a correct leader p_ℓ that has successfully obtained a verifiable vector. Observe that, under IC, the verifiable vector is directly admissible as the decision. This leads to a trivial persuasion protocol for IC. First, p_ℓ broadcasts the verifiable vector. Upon receiving the vector, a correct process stores the vector and sends back to p_ℓ a partial signature for the digest d of the vector. Then, p_ℓ can aggregate the partial signatures into a $(t+1)$-threshold signature. Importantly, this threshold signature satisfies the definition of a PoD for d, as (1) it must be signed by at least $t+1-f$ correct processes that have stored the pre-image (the vector), and (2) the pre-image is admissible according to IC. Thus, p_ℓ can conclude the persuasion by distributing d and its PoD to all processes. This eventually yields a

[3]Interestingly, this deviates from a common pattern in the leader-based paradigm in which success is guaranteed at the first view with a correct leader.

dispersal protocol that exchanges $O(nL_o + n(f+1)\kappa)$ bits. Furthermore, the full composition under ADA-DARE implies an IC protocol with the same communication complexity, which is word-optimal as $L_o = nL_{in}$ and $\Omega(n^2 L_{in})$ is inevitable for IC.

Efficient persuasion for strong & external validity (SMVBA). A keen reader might notice that, if VVC with a value (and signature) as its entry is realized, we can use a SNARK [46] to build a succinct proof of whether a value v is admissible or not (using our example in the VVC intuition). Thus, by distributing v and this proof, a leader can persuade everyone to build a PoD on the digest of v. However, SNARK is known to be computationally costly to compute. Is it still possible to efficiently perform persuasion with a weaker cryptographic tool? We answer this affirmatively by combining the idea of Big Buckets ([67]) and Multiverse Threshold Signature Scheme (MTSS) ([6, 41]). In a nutshell, MTSS is a cryptographic primitive that is weaker than SNARK and has two important features: (1) dynamic *universe* (set of processes) selection and (2) dynamic threshold. Moreover, the choice of universe and threshold can be done without additional communication.

First, to avoid the $\Omega(fnL_o)$ factor in the communication complexity, VVC is executed over digests as entries. Suppose that a correct leader p_ℓ has successfully obtained a verifiable vector. If there is a digest d that is signed by $t+1$ processes, then p_ℓ can make a trivial 'positive certificate' by combining them into a $(t+1)$-threshold signature. The positive certificate proves that (1) the pre-image of d has been observed by $t+1-f$ correct processes, and (2) the pre-image is externally valid. Using the positive certificate, p_ℓ is able to persuade other processes to build a PoD for a digest d, in a similar fashion to persuasion for IC.

However, what if no such digest exists? In this case, using the idea of Big Buckets, p_ℓ can divide the domain of the digest into a constant number ($O(1)$) of *buckets* (intervals) such that each bucket contains at most $n-(t+1)=t$ entries from the vector. p_ℓ then broadcasts the buckets, and upon their reception, a correct process signs every bucket to which its entry does *not* belong. Then, using MTSS and the verifiable vector, for each bucket, p_ℓ will be able to build a 'bucket negative certificate' that proves that some correct processes' digest is not included in that bucket. In a nutshell, it will consist of (1) at most $f_1 \leq f$ PoM, (2) at most $f_2 \leq f - f_1$ entries (from faulty processes), and (3) an $(t+1-f_1-f_2)$-MTSS where the universe consists of all processes except those in (1) and (2). Therefore, a bucket negative certificate consists of $O(f\kappa)$ bits. As there are $O(1)$ buckets, a 'negative certificate', testifying that there is no common proposal from the correct processes, can be built using only $O(\kappa f)$ bits. Using this negative certificate, p_ℓ can persuade all processes that its input, which must be externally valid, is admissible. Similarly, p_ℓ will be able to obtain and distribute a digest d along with a PoD for d.

The idea eventually yields a dispersal protocol that exchanges $O(nL_o+n(f+1)\kappa)$ bits. Furthermore, the full composition under ADA-DARE implies an SMVBA protocol with the same communication complexity, which is word-optimal.

Summary. In brief, we propose ADA-DARE: a new efficient generic technique to achieve BA for any validity property in the honest majority regime ($n = 2t+1$). The adaptive Retrieve phase is achieved by a novel, though simple, use of an expander graph combined with erasure codes and cryptographic accumulators. The Dispersion is reduced to two problems: (i) verifiable vector collection (VVC) and (ii) persuasion. VCC is adaptively realized via an iterative use of the introduced LearnOrExpose primitive, which forces a non-participating process to share its entry or be exposed. Then, we provide an efficient persuasion protocol for both IC and SMVBA. The full composition yields the optimal $O(n(L_o+f))$ word complexity for both SMVBA and IC, arguably at the very top of the hierarchy of the Byzantine Agreement problems.

3 SYSTEM MODEL & PRELIMINARIES

Processes. We consider a static system $\Pi = \{p_1, p_2, ..., p_n\}$ of $n = 2t+1$ processes, for some $t > 0$. At most t processes can be Byzantine: these processes can behave arbitrarily. If a process is Byzantine, we say that the process is *faulty*; otherwise, we say that the process is *correct* (or *honest*). We denote by $0 \leq f \leq t$ the *actual* number of faulty processes. Lastly, we assume that the faulty processes are computationally bounded, i.e., they cannot forge signatures of correct processes.

Synchronous environment. Processes communicate by exchanging messages over a complete authenticated synchronous network: (1) the receiver of a message is aware of the sender's identity, (2) every pair of correct processes can exchange messages directly, and (3) if a correct process sends a message to a correct process, the message is received after at most Δ time units, where Δ is known by every correct process. The existence of the known bound on message delays allows us to design protocols in a round-based paradigm: if a correct process sends a message to another correct process at the beginning of some round, the message is received by the end of the same round.

Values. We denote by Value_I the set of L_{in}-bit values processes can propose to ADA-DARE. Moreover, Value_O denotes the set of L_o-bit values processes can decide from ADA-DARE. Finally, for every correct process p_i, $\mathsf{input}(p_i)$ denotes p_i's proposal to ADA-DARE.

Reed-Solomon codes [68]. ADA-DARE relies on Reed-Solomon (RS) erasure codes (no error correction is required). We denote by $\mathsf{encode}(\cdot)$ and $\mathsf{decode}(\cdot)$ the RS' encoding and decoding algorithms, respectively. In a nutshell, $\mathsf{encode}(\cdot)$ takes a value v, chunks it into the coefficients of a polynomial $P_v(\cdot)$ of degree t (the maximum number of faults), and outputs n (the total number of processes) evaluations of the polynomial (RS symbols): $\mathsf{encode}(v) = [P_v(1), ..., P_v(n)]$. $\mathsf{decode}(\cdot)$ takes a set of $t+1$ RS symbols S and interpolates them into a polynomial of degree t, whose coefficients are concatenated and output. The size of one RS symbol of an L-bit value is $O\big(\max(\frac{L}{t+1}, \log n)\big)$.

Cryptographic accumulators [36, 50, 63]. Cryptographic accumulators are tools for aggregating a set of values into a single accumulation value, while also generating a verifiable witness for each value in the set. This procedure allows any process to validate the presence of a value in the set using its corresponding witness and the accumulation value. The accumulator scheme is assumed to be collision-resistant. Concretely, this paper utilizes the bilinear accumulator whose witness has a size of $O(\kappa)$ bits. We relegate the formal definition of cryptographic accumulators to Appendix A.

Digests. We design a collision-resistant Digest function, where $\mathsf{Digest}(\cdot) : \mathsf{Value}_I \rightarrow \mathsf{Digest} \triangleq \{0,1\}^{\kappa}$. Concretely, the Digest function takes a value v as input and performs the following steps: (1) It encodes v into $[P_v(1), P_v(2), ..., P_v(n)]$ using Reed-Solomon codes. (2) It aggregates $[P_v(1), P_v(2), ..., P_v(n)]$ into a value z_v using the bilinear accumulator. (3) It returns z_v. We order all digests according to the lexicographic order; d_{min} (resp., d_{max}) denotes the minimum (resp., maximum) digest. The formal definition of digests can be found in Appendix A.

Threshold signatures [58, 70]. ADA-DARE relies on a (k, n)-threshold signature scheme [58, 70] with $k = t+1$. In a (k, n)-threshold signature scheme, each process holds a distinct private key; there exists a single public key. Each process p_i can use its private key to produce a partial signature of a message m by invoking $\mathsf{ShareSign}_i^k(m)$. A partial signature *psignature* of a message m produced by process p_i can be verified by $\mathsf{ShareVerify}_i^k(m, \mathit{psignature})$. Lastly, a set $S = \{\mathit{psignature}_i\}$ of partial signatures, where $|S| = k$ and, for each $\mathit{psignature}_i \in S$, $\mathit{psignature}_i = \mathsf{ShareSign}_i^k(m)$, can be combined into a *single* (threshold) signature by invoking $\mathsf{Combine}^k(S)$; a threshold signature *tsignature* of message m can be verified with $\mathsf{CombinedVerify}^k(m, \mathit{tsignature})$. Where appropriate, invocations of $\mathsf{ShareVerify}^k(\cdot)$ and $\mathsf{CombinedVerify}^k(\cdot)$ are implicit in our descriptions. We denote by P_Signature and T_Signature a partial signature and a (combined) threshold signature, respectively. Importantly, under a security parameter κ, each signature is of size $O(\kappa)$ bits.

Multiverse threshold signatures [6, 41]. In the persuade sub-protocol (Algorithm 8 analyzed in Appendix C.6) used by the instance of ADA-DARE dedicated to strong & external validity, we rely on a modern implementation [55, 65] of a multiverse threshold signature scheme (MTSS) [6, 41]. Intuitively, the MTSS behaves as a classic TSS, but enjoys: (1) a transparent setup (i.e., an established PKI and a common random string[4], (2) a dynamic threshold, and (3) a communication-free universe setup to dynamically choose the universe, i.e., the exact subset of processes allowed to contribute to the threshold. We relegate the formal definition of MTSS to Appendix A.

Communication complexity of synchronous protocols Let P be any synchronous protocol and let $execs(\mathsf{P})$ denote the set of all possible executions of P.[5] Let $\alpha \in execs(\mathsf{P})$ be an execution of P. A word is a string of $O(\kappa)$ bits. We define the *bit* (resp., *word*) *complexity* of α as the number of bits (resp., words) sent in messages by all correct processes in α. The bit (resp., word) complexity of P is then defined as the maximum bit (resp., word) complexity over $execs(P)$.

4 ADA-DARE

This section fixes some generic validity val, and presents ADA-DARE$_{\mathsf{val}}$ (ADAptively Disperse, Agree, REtrieve), which is composed of three algorithms:

(1) ADA-DISP$_{\mathsf{val}}$, which disperses the proposals, valid for the specific validity property val;
(2) **CKS** [32], which ensures agreement on the digest of a previously dispersed proposal; and
(3) ADA-RETRIEVE, which rebuilds the proposal corresponding to the agreed-upon digest.

As explained in §2, ADA-DARE$_{\mathsf{val}}$ closely follows the structure of previous (and well-established) "Disperse, Agree, Retrieve" compositions [23]. A full formal analysis of the composition itself, including an algorithm and proofs of correctness and complexity, can be found in Appendix B. Here, we present the interface, properties, and complexity of each of the three main sub-components, of which dispersal (ADA-DISP$_{\mathsf{val}}$) and retrieval (ADA-RETRIEVE) form the core challenge and contribution.

4.1 ADA-DISP$_{\mathsf{val}}$

Interface & properties. In a nutshell, correct processes aim to collectively disperse some valid value $v \in \mathsf{val}(c)$, for the specific validity property val and the input configuration[6] c: eventually, all correct processes acquire a digest along with a proof-of-dispersal (PoD), a threshold signature that proves the pre-image value of the digest is valid and has been successfully dispersed. Concretely, ADA-DISP$_{\mathsf{val}}$ exposes the following interface:

- **request** propose($v \in$ Value): a process proposes a value v; each correct process invokes propose(v) exactly with externallyValid(v) = *true*[7].
- **indication** acquire($d \in$ Digest_Value, $\sigma^d \in$ T_Signature): a process acquires a pair (d, σ^d).

We say that a correct process *obtains* a threshold signature (resp., a value) if and only if it stores the signature (resp., the value) in its local memory. (Obtained values can later be retrieved by all correct processes using ADA-RETRIEVE; see §4.3 and Algorithm 3.) ADA-DISP$_{\mathsf{val}}$ ensures the following:

- *Integrity:* If a correct process triggers acquire(d, σ^d), then CombinedVerify$^{t+1}(\langle \text{ACK}, d\rangle, \sigma^d)$) = *true.*
- *Termination:* Every correct process eventually acquires at least one digest-signature pair.
- *Redundancy:* Let a correct process obtain (d, σ^d) such that CombinedVerify$^{t+1}(\langle \text{ACK}, d\rangle, \sigma^d)$ = *true*, for some digest d and some threshold signature σ^d. Then, (at least) $t + 1 - f$ correct processes have obtained a value v such that (1) Digest$(v) = d$, and (2) $v \in \mathsf{val}(c)$ for the

[4]Different from a common reference string such that q-SDH public parameters [29, 35].
[5]We omit the negligible subset of executions where cryptographic properties are violated.
[6]See Appendix C.1 for the definition of input configuration.
[7]externallyValid$(\cdot)$ is the globally verifiable predicate employed by external validity.

input configuration c defined by the correct proposals. Let us remark that (2) might imply $\mathsf{externallyValid}(v) = \mathit{true}$ depending on val.

Note that it is not required for all correct processes to acquire the same digest value (nor the same threshold signature). Moreover, the specification allows for multiple acquired pairs.

Complexity. Both ADA-DISP$_{\mathsf{val}_{ic}}$ and ADA-DISP$_{\mathsf{val}_{sv}}$ exchange $O(nL_o + n(f+1)\kappa)$ bits, where val_{ic} and val_{sv} stand for the validity properties of IC and SMVBA, respectively. Also, every correct process acquires at least one digest-signature pair by $O(f)$ rounds. Finally, all correct processes halt simultaneously within $O(n)$ rounds.

Implementation. The details on ADA-DISP's implementation are relegated to §6.

4.2 CKS

Interface & properties. CKS is a VBA algorithm for some generic $\mathsf{valid}_{\mathbf{CKS}}(\cdot)$ predicate.[8] In **CKS**, processes propose and decide pairs ($d \in \mathsf{Digest\_Value}, \sigma_d \in \mathsf{T\_Signature}$); moreover, $\mathsf{valid}_{\mathbf{CKS}}(d, \sigma^d) \equiv \mathsf{CombinedVerify}^{t+1}(\langle \text{ACK}, d\rangle, \sigma^d)$. In other words, a valid value must be in the form of a digest accompanied by its PoD.

Complexity. CKS achieves $O(\kappa n(f+1))$ bit complexity for $O(\kappa)$-bits values and correct processes decide in $O(n)$ rounds.

Implementation. In a nutshell, **CKS** is a validated BA protocol that follows the silent views paradigm. Under the hood, it employs a fallback protocol in a scenario when there are too many faulty processes and thus, unable to decide in its optimistic path. Turning the fallback protocol [61] into a validated BA protocol is straightforward, by ignoring each received message containing a non-externally valid value (details are given in Appendix D). Moreover, **CKS** incorporates the correction of a minor technical flaw in [32] addressed by [38].

4.3 ADA-RETRIEVE

Interface & properties. In ADA-RETRIEVE, each correct process starts with (1) a digest and (2) either (a) some corresponding pre-image value, or (b) $\perp$. Eventually, all correct processes output the same value (the pre-image). Formally, ADA-RETRIEVE exposes the following interface:

- **request** $\mathsf{input}(d \in \mathsf{Digest\_Value}, v \in \mathsf{Value} \cup \{\perp\})$: a process inputs a digest d and either $\perp$ or a value v such that $\mathsf{Digest}(v) = d$; each correct process invokes $\mathsf{input}(\cdot)$ exactly once[9]. Moreover, the following is assumed:
  - No two correct processes invoke $\mathsf{input}(d_1, v_1)$ and $\mathsf{input}(d_2, v_2)$ with $d_1 \neq d_2$.
  - At least $t+1-f$ correct processes invoke $\mathsf{input}(d, v)$ with $v \neq \perp$ (i.e., $\mathsf{Digest}(v) = d$).
- **indication** $\mathsf{output}(v' \in \mathsf{Value})$: a process outputs a value v'.

The following properties are ensured:

- *Agreement:* No two correct processes output different values.
- *Validity:* Let a correct process input a value v. No correct process outputs a value $v' \neq v$.
- *Termination:* Every correct process eventually outputs a value.

Complexity. ADA-RETRIEVE exchanges $O(nL_o+n(f+1)\kappa)$ bits. Moreover, ADA-RETRIEVE terminates in $O(\log n)$ time.

Implementation. The details on ADA-RETRIEVE's implementation are relegated to §5.

[8] Recall that the interface and properties of Byzantine consensus algorithms are introduced in §1.
[9] We underline that $d \neq \perp$ even if $v = \perp$.

5 ADA-RETRIEVE

This section presents ADA-RETRIEVE, which improves upon the erasure code-based reconstruction from [62] by adding a preceding phase that ensures only $O(f)$ correct processes do not receive the pre-image value (v) yet. We present the implementation in §5.1 and its analysis in §5.2. Full details are relegated to Appendix E.

5.1 Implementation

Algorithm 1 ADA-RETRIEVE: Pseudocode for process p_i

```
 1: Uses:
 2:     ExpanderGraph, instance $G$                                         ▷ see [73]
 3: Input parameters:
 4:     Digest_Value $digest_i \leftarrow d$                                ▷ the input $d$
 5:     Value $v_i \leftarrow v$                  ▷ the input $v$. if $v \neq \bot$, then $d = \text{Digest}(v)$
 6: Local variables:
 7:     RS_Symbol $symbol_i \leftarrow \bot$           ▷ the $i$-th RS symbol from the pre-image
 8:     Witness $witness_i \leftarrow \bot$                        ▷ witness for $symbol_i$
 9: for Integer $k \leftarrow 1$ to $O(\log n)$:                          ▷ to almost everywhere
10:     if $v_i \neq \bot$ and has not gossiped: multicast ⟨GOSSIP, $v_i$⟩ to $i$'s neighbour in $G$
11:     if received ⟨GOSSIP, Value $v'$⟩, $\text{Digest}(v') = digest_i$, and $v_i = \bot$: $v_i \leftarrow v'$
12: if $v_i = \bot$: broadcast ⟨COMPLAIN⟩                                ▷ to everywhere
13: else: $symbol_i \leftarrow P_{v_i}(i)$; $witness_i \leftarrow \text{CreateWit}(ak, digest_i, (i, symbol_i))$
14: if $v_i \neq \bot$, upon receiving ⟨COMPLAIN⟩ from $p_j$:
15:     let $s_j \leftarrow P_{v_i}(j)$ and $w_j \leftarrow \text{CreateWit}(ak, digest_i, (j, s_j))$
16:     send ⟨YOUR_SYMBOL, $s_j$, $w_j$⟩ to $p_j$
17: if $v_i = \bot$, upon receiving ⟨YOUR_SYMBOL, RS_Symbol $s$, Witness $w$⟩ and $\text{Verify}(ak, digest_i, w, (i, s)) = true$:
18:     $symbol_i \leftarrow s$; $witness_i \leftarrow w$
19:     broadcast ⟨REQ_SYMBOLS⟩
20: upon receiving ⟨REQ_SYMBOLS⟩ from $p_j$: send ⟨MY_SYMBOL, $symbol_i$, $witness_i$⟩ to $p_j$
21: if $v_i \neq \bot$: trigger output($v_i$)
22: else, upon receiving ⟨MY_SYMBOL, RS_Symbol $s_j$, Witness $w_j$⟩ from $t + 1$ different $p_j$ with
23: $\text{Verify}(ak, digest_i, w_j, (j, s_j)) = true$:
24:     reconstruct $v$ from $t + 1$ RS via decode($\cdot$)
25:     trigger output($v$)
```

We present the pseudocode in Algorithm 1. At a high level, ADA-RETRIEVE consists of two phases: (1) *to almost everywhere* phase that ensures all but $O(f)$ correct processes received the pre-image value, and (2) *to everywhere* that ensures all correct processes received the pre-image value. Let us explain the idea of the first phase. Under the hood, ADA-RETRIEVE uses an expander graph during the first phase. In a nutshell, an expander graph is a (typically, low-degree) graph with a good expansion property. Roughly speaking, it means each vertex has a few neighbours, and any "relatively small" set of vertices has many neighbours. In ADA-RETRIEVE, we use a specific expander graph from [73], which has some nice additional properties as shown in the following lemma.

Lemma 1. (Restated from [73]) For all n, there exists a constant $d > 0$ that is independent of n and some d-regular graph with n vertices, where $\forall T \subset V$ with $|T| \leq n/72$, there is a connected component $P(T) \in V \setminus T$ of size at least $n - 6|T|$. Furthermore, the diameter of $P(T)$ is $O(\log n)$.

We underline that the expander graph can be precomputed, given to all processes before the protocol starts, and reused across multiple instances of the ADA-RETRIEVE.

ADA-RETRIEVE gossips the pre-image value for $O(\log n)$ rounds through the expander graph from Lemma 1 (line 9). By the pre-condition that $t + 1 - f$ correct processes input the pre-image value to ADA-Retrieve along with the guarantee from the graph, all but $O(f)$ correct processes will receive the pre-image value. Namely, it is ensured by the expander graph when $f \leq n/72$ while when $f > n/72$, $f \in \Omega(n)$ and therefore, the claim also holds even if $O(n)$ correct processes fail to obtain the pre-image during this phase. As a small note, unlike usual gossip, here a process only forwards a value to its neighbours at most once (line 10). Importantly, each correct process can validate the correctness of a value v' by comparing $\mathsf{Digest}(v')$ with the input digest d (line 11).

We now describe the *to almost everywhere phase*. Here, we exploit the fact that only $O(f)$ correct processes have not obtained the pre-image yet. Such process p_i first sends a broadcast to obtain the i-th RS symbol from $\mathsf{encode}(v)$ (line 12). Notice that a correct process p_j that has obtained v must be able to compute the i-th RS symbol along with an accumulator witness, proving the correctness of the RS symbol. Therefore, p_j will be able to reply p_i with the RS symbol and its witness (line 16). Importantly, p_i can also verify the correctness of the symbol via the witness. p_i then continue by collecting the other RS symbols from $\mathsf{encode}(v)$ through other processes (line 19). At this point, all correct processes p_j must have the j-th RS symbol along with its witness (either computing directly from the obtained v or received during the previous step). All correct processes p_j help p_i by sending their RS symbol share (line 20). Therefore, p_i must be able to collect at least $t + 1$ different RS symbols share, allowing reconstruction of v (line 24). Thus, all correct processes can output v, either by obtaining it directly or via reconstruction.

5.2 Analysis

Here we summarize the analysis; proofs are relegated to Appendix E.2 (Theorem 8 and Theorem 9).

Correctness. Validity follows from the fact that each correct process can validate the correctness of a value received during the *to almost everywhere* phase and is able to reconstruct v during the *to everywhere* phase. Agreement follows from the fact that each correct process can only output v. Finally, each correct process will output v and therefore, termination is guaranteed.

Complexities. In the *to almost everywhere* phase, each correct process sends at most one message containing $O(L_o + \kappa)$ bits to a constant number of processes. Then, in *to everywhere* phase, each correct process sends to $O(f)$ faulty processes and $O(f)$ correct processes that do not have v yet a message of size $O(\max(\frac{L_o}{t+1}, \log n) + \kappa)$. Therefore, the total bits sent will be $O(nL_o + n(f + 1)\kappa)$. The latency follows from the fact that *to almost everywhere* phase consists of $O(\log n)$ rounds and *to everywhere* phase consists of $O(1)$ rounds.

6 ADA-DISP

In this section, we present ADA-Disp$_{\mathsf{val}}$ (see Algorithm 2), a dispersal protocol for a specified validity property val. Recall that the goal of dispersion is to enable every correct process to produce a digest d and a threshold signature σ^d proving that at least $t + 1 - f$ correct processes have obtained a valid value v with $\mathsf{Digest}(v) = d$. Thus, any such digest output by ADA-Disp$_{\mathsf{val}}$ can be decided upon as the value that corresponds to that digest can be retrieved. The key challenge in solving the dispersion problem is ensuring that the dispersed value satisfies the (stronger) validity condition.[10] The full detail of ADA-Disp$_{\mathsf{val}}$ is relegated to Appendix C.

The dispersal protocol proceeds in views, each with a unique leader; each view contains a constant number of rounds, and the entire ADA-Disp$_{\mathsf{val}}$ protocol terminates within $O(f)$ rounds. Each view consists of two phases: (i) catch-up: if a proof-of-dispersal (PoD) has already been created in a previous view, the leader assists in its dissemination; (ii) attempt: the leader attempts to procure sufficient information on input values to certify a valid value.

[10]For weak validity (resp., external) validity, the leader can propose its value (resp., any valid value).

The catch-up phase (invoked at line 9) unfolds as follows: any process that has not yet obtained a PoD can request it from the leader, which shares its PoD (if previously obtained). An analysis of the catch-up phase is relegated to Appendix C.2 (see Algorithm 4). In brief, (1) if a correct process possesses a PoD and initiates the catch-up phase under a correct leader, every correct process acquires a PoD within the same view (see Lemma 11), and (2) the total bit complexity of all the views executed after all correct processes have obtained a PoD is $O(\kappa n f)$ (see Lemma 14). During the attempt phase (line 11), there are two possibilities: (a) if the leader detects a failure, the LearnOrExpose protocol is executed (line 25); (b) if the leader signals a success, the persuade protocol is executed (line 23), during which the (correct) leader constructs and disseminates a PoD. The series of attempt phases before the first correct leader signals a success allows for the verifiable vector collection, which we discuss further in the following subsection.

6.1 Verifiable Vector Collection

The goal of verifiable vector collection is for the leader to assemble a vector of entries for each process, where an entry is either (i) the (digest of the) input value for that process, or (ii) a proof that the process is malicious. For strong validity, the vector contains only digests; for interactive consistency, it contains the values themselves—in this section, we will refer to the entry as the (digest of the) input value to reflect this ambiguity.

The verifiable vector collection is executed over a series of views until some honest leader succeeds. In each view with an honest leader, either (i) the protocol completes, (ii) at least one new value (from a malicious process) is disclosed to every process, or (iii) a proof-of-misbehavior (PoM) is produced for a new malicious process and distributed to everyone. After (at most) f views with honest leaders, every malicious process has either disclosed its value or been proved malicious.

When a view begins, each process that has not yet received a PoD sends the signed (digest of their) input value to the leader. At this point, the leader has the required entry information for all $n - f$ processes, along with any PoMs and disclosed values from previous views. If the leader has all n entries, then the verifiable vector collection completes.

Otherwise, the leader initiates the LearnOrExpose protocol (Algorithm 6 in Appendix C.3) targeting a single (malicious) process that did not send its (digest of its) input value. The LearnOrExpose primitive compels a process under suspicion to either publicly disclose its signed entry, or face the consequences of not doing so—the generation and dissemination of an undeniable PoM. First, the leader disseminates the target identity. All processes then request the entry information from the target. If any correct process receives such (signed) entry information from the target, it forwards it to the leader. Otherwise, it sends a signed accusation message to the leader. If the leader receives an entry signed by the target, it discloses this information to everyone. Otherwise, it assembles $t + 1$ signed accusations using a threshold signature into a PoM, which is then broadcast.

Importantly, within $O(f)$ views, some (correct) leader receives a verifiable vector. The following lemma is proved in Appendix C.7.3 (see Lemma 30).

Lemma 2 (Verifiable Vector Collection)**.** *If not all the correct processes acquired a digest-signature pair $(d, \mathrm{PoD}(d))$ by view $2f$, then* ADA-DISP$_{\mathrm{val}}$ *achieves verifiable vector collection at view $\ell = 2f+1$ under (a correct leader) p_ℓ.*

PROOF SKETCH. If a correct process acquires a well-formed PoD in some view, this PoD is forwarded to all processes by the first correct leader of a subsequent view. Assume that no correct process acquires a well-formed PoD by view $2f$. Then f correct leaders successively led some views without success. In each of these f views, the size of the input materials related to faulty processes is incremented by 1. Hence, by the end of $2f$ views, the size of the input materials related to faulty processes equals at least f. Thus, the next correct leader $p_{\ell \leq 2f+1}$ collects a verifiable vector. □

Algorithm 2 ADA-DISP$_{\mathsf{val}}$($start\_value_i$): Simplified pseudocode (for process p_i)
(c.f., Algorithm 5 in Appendix C.2 and Algorithm 11 in Appendix C.7 for a detailed presentation)

```
1: Input Parameters:
2:     Value $start\_value_i$                                              ▷ input($p_i$)
3: Variables:
4:     Digest_Value $certified\_digest_i \leftarrow \bot$
5:     PoD $pod_i \leftarrow \bot$                                          ▷ succinct proof of dispersal
6:     Structure $material_i \leftarrow \emptyset$   ▷ (informal) includes entries and PoM indirectly obtained through LearnOrExpose
7:     Entry $entry_i \leftarrow \mathsf{entry}_{\mathsf{val}}(start\_value_i)$   ▷ The entry is either the proposal or its digest, depending on val
8: for $\ell = 1$ to $n$:
9:     $certified\_digest_i, pod_i \leftarrow \mathsf{catch\_up}(p_\ell, certified\_digest_i, pod_i)$   ▷ see Algorithm 4 in Appendix C.2 (4 rounds)
10:    if $pod_i = \bot$:                                                  ▷ $pod_i = \bot$ initiates the attempt phase
11:        ▷ Attempt Phase
12:        Round 1:
13:            send ⟨DISCLOSE, $entry_i$, $\mathsf{ShareSign}_i^{t+1}$(DISCLOSE, $entry_i$)⟩ to $p_\ell$   ▷ inform the leader of its entry
14:        Round 2:                                                        ▷ executed only by $p_\ell$
15:            if $p_i = p_\ell$:
16:                update $material_i$ with the received DISCLOSE messages
17:                if $material_i$ contains, for each process, a signed entry, or a PoM:   ▷ check if VCC has been achieved
18:                    broadcast ⟨SUCCESS⟩
19:                else:                                                   ▷ Select a new process to accuse
20:                    broadcast ⟨FAIL, $p_a$⟩ with $p_a$ absent from $material_i$
21:        Round 3:
22:            if ⟨SUCCESS⟩ is received from $p_\ell$:
23:                $certified\_digest_i, pod_i \leftarrow$ outputs of $\mathsf{persuade}_{\mathsf{val}}(start\_value_i, p_\ell, material_i)$
24:            else if ⟨FAIL, $p_a$⟩ is received from $p_\ell$:            ▷ Collect one more disclose message for the next view
25:                $material_i \leftarrow$ outputs of $\mathsf{LearnOrExpose}_{\mathsf{val}}(start\_value_i, p_\ell, p_a, material_i)$
26:    if $pod_i \neq \bot$: trigger acquire($certified\_digest_i, pod_i$)
```

In each view, there are $O(n)$ messages sent to/from the leader, and $O(n)$ messages sent to/from the LearnOrExpose target. (A correct process never sends its value more than once to any other process, thus ensuring that malicious processes cannot yield more than $O(nf)$ messages via malicious requests.) The following lemma is proved in Appendix C.9 (see Lemma 35).

Lemma 3. Let val be a validity property. Let L_e be the maximum between an entry size (i.e., the digest or the input value of a process depending on val) and κ. Let the communication complexity of $\mathsf{persuade}_{\mathsf{val}}$ under a Byzantine leader be $O(\kappa n)$. Then, ADA-DISP$_{\mathsf{val}}$ exchanges $O(nf(L_e + \kappa))$ bits to achieve verifiable vector collection.

PROOF SKETCH. The $n(f+1)L_e$ term comes from the sending of entries to(at most) $2f+1$ leaders and (at most) f Byzantine processes (via the LearnOrExpose primitive). The careful bookkeeping of previous LearnOrExpose interactions limits the communication complexity related to the sending of entries to $O(n(f+1)L_e)$. The $n(f+1)\kappa$ term comes from (1) Lemma 2, (2) the leader-based communication pattern, and (3) the $O(\kappa)$-bit size of the messages, excluding entry messages. □

6.2 Persuasion

Persuasion begins once a correct leader obtains a verifiable vector containing signed entires and PoMs. The leader's responsibility is to choose a value/digest to disperse and persuade enough processes to accept that value/digest, concluding the dispersion problem. Any digest and signature output by the persuasion subprotocol should satisfy the requirements of the dispersion problem.

Interactive Consistency. Here, persuasion is straightforward. The leader broadcasts the vector of values, along with PoMs for the missing values. Each process responds with a signed acknowledgement. The leader assembles $t + 1$ of these signatures into a threshold signature, and broadcasts it (and a digest of the vector) as a PoD. All (correct) processes can ascertain that the vector is correct (via the signed values and PoMs) and has been obtained by at least $t + 1 - f$ correct processes.

Strong Validity. For strong validity, the process of persuasion is more sophisticated as the validity of a value must be proven. To this end, we introduce a new type of proof, a *bucket certificate*, which is a new feature of our approach, adapting the idea of [67]. There are two cases to consider.

In the easy case, the leader identifies at least $t + 1$ processes that have identical (signed) digests in the verifiable vector—and hence have the same input value. In this case, the leader can assemble those $t + 1$ signatures into a threshold signature, which we refer to as a *positive bucket certificate*; the leader then broadcasts the digest along with the bucket certificate as a PoD. The $t + 1$ signatures prove that the value is valid (i.e., if all honest processes started with the same value, then this must be that value), as well as show that the value is available to at least $t + 1 - f$ correct processes.

In the harder case, there is no such single digest supported by $t + 1$ processes. Thus, the leader can choose any value to disperse (e.g., its proposal). However, the leader needs to prove that not all correct processes proposed the same value (otherwise, strong validity could be violated). To this end, we introduce the idea of a *negative bucket certificate* that proves that not all correct processes started with the same value.

Concretely, the leader first examines the verifiable vector of entries and partitions the digests into $O(1)$ groups such that: (i) each group contains a range of digests, (ii) each group includes digests from at most t processes, and (iii) any two consecutive groups contain digests from at least $t + 1$ processes. These groups can be constructed greedily, and communicated by broadcasting $O(1)$ endpoints. Each process then sends to the leader a signed message indicating which groups its value *does not* belong to. The leader combines the signatures for each group into a single threshold signature certifying that a sufficient number of processes do not have their value in that range of digests. For each group, the leader then constructs a negative bucket certificate containing: (1) the (threshold) signature of processes that did not start with that range of digests; (2) the signed digests of (malicious) processes that are included in the verifiable vector but did not send a (proper) signature of a group; (3) the PoMs for the processes that did not have digests in the verifiable vector.

Notice that for each group, the total number of signatures plus the number of malicious signed digests plus the number of proofs-of-misbehavior must be at least $t + 1$ as no group contains digests supported by $t + 1$ processes in the verifiable vector. This proves that no digest in the group represents a value that was the initial value for every honest process.

After the leader has constructed a negative bucket certificate, it chooses any value[11], and disseminates it along with the bucket certificate. Then, the leader collects $t + 1$ signed acknowledgements (from all correct processes) and assembles them into a threshold signature. It then disseminates that signature (along with the digest) as a PoD.

One technical issue remains: unlike the "regular" threshold signatures used in this paper, the aforementioned approach requires the universe for the threshold signatures to not be fixed–it must exclude those processes that refuse to sign the groups or for which there exists a PoM (to avoid double counting). Thus, we rely on Multiverse Threshold Signature Scheme (MTSS) [6, 41] to construct the appropriate universe for the threshold signatures.

The following lemmas are proven in Appendix C.5, Appendix C.6, and Appendix C.9—see Lemma 20, Lemma 25, Lemma 38, and Lemma 36.

[11]If we want to satisfy external validity, it chooses a valid value.

Lemma 4. If protocol $\text{persuade}_{\text{val}_{ic}}$ (Algorithm 7) (resp., $\text{persuade}_{\text{val}_{sv}}$ (Algorithm 8)) is a subroutine of ADA-DISP$_{\text{val}_{ic}}$ (resp., ADA-DISP$_{\text{val}_{sv}}$), it solves the Persuasion problem parameterized with val_{ic} and $\text{VerifyVector}_{\text{val}_{ic}}$ (resp., val_{sv} and $\text{VerifyVector}_{\text{val}_{sv}}$).

Lemma 5. The execution of $\text{persuade}_{\text{val}_{sv}}$ (Algorithm 8) (resp., $\text{persuade}_{\text{val}_{ic}}$ (Algorithm 7)) under leader p_ℓ incurs an exchange of (1) $O(\kappa n)$ bits if p_ℓ is Byzantine, and (2) $O(nL_o + n(f+1)\kappa)$ bits (resp., $O(n^2(L_i + \kappa))$), otherwise.

Lemmas 2 to 5, alongside the exchange of at most $O(\kappa n f)$ bits following the attainment of a PoD by all correct processes (refer to Lemma 14 in Appendix C.2), lead to the complexity claimed in §4.1.

7 CONCLUSION

We have introduced ADA-DARE, a universal strategy for efficiently solving BA in the honest majority regime ($n > 2t$). Two specific instances achieve optimal word complexity for both SMVBA and IC.

REFERENCES

[1] Abd-El-Malek, M., Ganger, G. R., Goodson, G. R., Reiter, M. K., and Wylie, J. J. Fault-Scalable Byzantine Fault-Tolerant sServices. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005* (2005), A. Herbert and K. P. Birman, Eds., ACM, pp. 59–74.

[2] Abraham, I., Chan, T. H., Dolev, D., Nayak, K., Pass, R., Ren, L., and Shi, E. Communication complexity of byzantine agreement, revisited. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing* (2019), 317–326.

[3] Abraham, I., Malkhi, D., Nayak, K., Ren, L., and Spiegelman, A. Solida: A Blockchain Protocol Based on Reconfigurable Byzantine Consensus. In *21st International Conference on Principles of Distributed Systems, OPODIS 2017, Lisbon, Portugal, December 18-20, 2017* (2017), J. Aspnes, A. Bessani, P. Felber, and J. Leitão, Eds., vol. 95 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 25:1–25:19.

[4] Abraham, I., Malkhi, D., and Spiegelman, A. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019), pp. 337–346.

[5] Abraham, I., Nayak, K., and Shrestha, N. Communication and round efficient parallel broadcast protocols. *IACR Cryptol. ePrint Arch.* (2023), 1172.

[6] Baird, L., Garg, S., Jain, A., Mukherjee, P., Sinha, R., Wang, M., and Zhang, Y. Threshold signatures in the multiverse. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023* (2023), IEEE, pp. 1454–1470.

[7] Ben-Or, M., Goldwasser, S., and Wigderson, A. Completeness Theorems for Non-Cryptographic Fault-Tolerant Distributed Computation. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, O. Goldreich, Ed. ACM, 2019, pp. 351–371.

[8] Berman, P., Garay, J. A., and Perry, K. J. Bit Optimal Distributed Consensus. In *Computer science: research and applications*. Springer, 1992, pp. 313–321.

[9] Bhangale, A., Liu-Zhang, C. D., Loss, J., and Nayak, K. Efficient Adaptively-Secure Byzantine Agreement for Long Messages. In *Advances in Cryptology - ASIACRYPT - 28th International Conference on the Theory and Application of Cryptology and Information Security* (Taipei, Taiwan, 2022), vol. 13791 LNCS, pp. 504–525.

[10] Blum, E., Boyle, E., Cohen, R., and Liu-Zhang, C.-D. Communication Lower Bounds for Cryptographic Broadcast Protocols. In *37th International Symposium on Distributed Computing (DISC)* (L'Aquila, Italy, 2023), pp. 10:1—-10:19.

[11] Blum, E., Katz, J., Liu-Zhang, C., and Loss, J. Asynchronous byzantine agreement with subquadratic communication. In *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part I* (2020), R. Pass and K. Pietrzak, Eds., vol. 12550 of *Lecture Notes in Computer Science*, Springer, pp. 353–380.

[12] Boneh, D., Drijvers, M., and Neven, G. Compact multi-signatures for smaller blockchains. In *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II* (2018), T. Peyrin and S. D. Galbraith, Eds., vol. 11273 of *Lecture Notes in Computer Science*, Springer, pp. 435–464.

[13] Boyle, E., Cohen, R., and Goel, A. Breaking the O($\sqrt{}$ n)-bit barrier: Byzantine agreement with polylog bits per party. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing* (2021), 319–330.

[14] Boyle, E., Jain, A., Prabhakaran, M., and Yu, C. H. The bottleneck complexity of secure multiparty computation. In *45th International Colloquium on Automata, Languages, and Programming, (ICALP)* (Prague, Czech Republic, 2018), vol. 107, pp. 1–16.

[15] Buchman, E. *Tendermint: Byzantine Fault Tolerance in the Age of Blockchains.* PhD thesis, University of Guelph, 2016.

[16] Buchman, E., Kwon, J., and Milosevic, Z. The latest gossip on bft consensus. Tech. Rep. 1807.04938, arXiv, 2019.

[17] Canetti, R. Universally composable signature, certification, and authentication. In *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004), 28-30 June 2004, Pacific Grove, CA, USA* (2004), IEEE Computer Society, p. 219.

[18] Castro, M., and Liskov, B. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems 20*, 4 (2002).

[19] Chan, T. H., Pass, R., and Shi, E. Sublinear-Round Byzantine Agreement Under Corrupt Majority. In *Public-Key Cryptography* (Edinburgh, UK, 2020), vol. 12111 LNCS, pp. 246–265.

[20] Chen, J., and Micali, S. Algorand: A secure and efficient distributed ledger. *Theoretical Computer Science 777* (2019), 155–183.

[21] Chlebus, B. S., Kowalski, D. R., and Olkowski, J. Deterministic fault-tolerant distributed computing in linear time and communication. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC 2023, Orlando, FL, USA, June 19-23, 2023* (2023), R. Oshman, A. Nolin, M. M. Halldórsson, and A. Balliu, Eds., ACM, pp. 344–354.

[22] Civit, P., Dzulfikar, M. A., Gilbert, S., Gramoli, V., Guerraoui, R., Komatovic, J., and Vidigueira, M. Byzantine Consensus Is Θ (n^2): The Dolev-Reischuk Bound Is Tight Even in Partial Synchrony! In *36th International Symposium on Distributed Computing (DISC 2022)* (2022), Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[23] Civit, P., Gilbert, S., Guerraoui, R., Komatovic, J., Monti, M., and Vidigueira, M. Every bit counts in consensus. In *37th International Symposium on Distributed Computing, DISC 2023, October 10-12, 2023, L'Aquila, Italy* (2023), R. Oshman, Ed., vol. 281 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 13:1–13:26.

[24] Civit, P., Gilbert, S., Guerraoui, R., Komatovic, J., Paramonov, A., and Vidigueira, M. All byzantine agreement problems are expensive. *CoRR abs/2311.08060* (2023).

[25] Civit, P., Gilbert, S., Guerraoui, R., Komatovic, J., and Vidigueira, M. On the Validity of Consensus. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC 2023, Orlando, FL, USA, June 19-23, 2023* (2023), R. Oshman, A. Nolin, M. M. Halldórsson, and A. Balliu, Eds., ACM, pp. 332–343.

[26] Civit, P., Gilbert, S., Guerraoui, R., Komatovic, J., and Vidigueira, M. On the validity of consensus. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing, PODC 2023, Orlando, FL, USA, June 19-23, 2023* (2023), R. Oshman, A. Nolin, M. M. Halldórsson, and A. Balliu, Eds., ACM, pp. 332–343.

[27] Civit, P., Gilbert, S., Guerraoui, R., Komatovic, J., and Vidigueira, M. Strong Byzantine Agreement with Adaptive Word Complexity. *arXiv preprint arXiv:2308.03524* (2023).

[28] Coan, B. A., and Welch, J. L. Modular Construction of a Byzantine Agreement Protocol with Optimal Message Bit Complexity. *Inf. Comput. 97*, 1 (1992), 61–85.

[29] Cohen, R., Doerner, J., Kondi, Y., and Shelat, A. Guaranteed output in $o(\sqrt{n})$ rounds for round-robin sampling protocols. In *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part I* (2022), O. Dunkelman and S. Dziembowski, Eds., vol. 13275 of *Lecture Notes in Computer Science*, Springer, pp. 241–271.

[30] Cohen, S., Goren, G., Kokoris-Kogias, L., Sonnino, A., and Spiegelman, A. Proof of availability and retrieval in a modular blockchain architecture. In *Financial Cryptography and Data Security - 27th International Conference, FC 2023, Bol, Brač, Croatia, May 1-5, 2023, Revised Selected Papers, Part II* (2023), F. Baldimtsi and C. Cachin, Eds., vol. 13951 of *Lecture Notes in Computer Science*, Springer, pp. 36–53.

[31] Cohen, S., Keidar, I., and Spiegelman, A. Not a coincidence: Sub-quadratic asynchronous byzantine agreement WHP. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference* (2020), H. Attiya, Ed., vol. 179 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 25:1–25:17.

[32] Cohen, S., Keidar, I., and Spiegelman, A. Make every word count: Adaptive byzantine agreement with fewer words. In *26th International Conference on Principles of Distributed Systems (OPODIS 2022)* (2023), Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[33] Correia, M. From Byzantine Consensus to Blockchain Consensus. *Essentials of Blockchain Technology 41* (2019), 2019.

[34] Crain, T., Gramoli, V., Larrea, M., and Raynal, M. DBFT: Efficient Leaderless Byzantine Consensus and its Applications to Blockchains. In *Proceedings of the 17th IEEE International Symposium on Network Computing and Applications (NCA'18)* (2018), IEEE.

[35] Das, S., Xiang, Z., and Ren, L. Powers of tau in asynchrony. *IACR Cryptol. ePrint Arch.* (2022), 1683.

[36] Derler, D., Hanser, C., and Slamanig, D. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In *Topics in Cryptology - CT-RSA 2015, The Cryptographer's Track at the RSA Conference 2015, San Francisco, CA, USA, April 20-24, 2015. Proceedings* (2015), K. Nyberg, Ed., vol. 9048 of *Lecture Notes in Computer Science*, Springer, pp. 127–144.

[37] Dolev, D., and Reischuk, R. Bounds on information exchange for Byzantine agreement. *Journal of the ACM (JACM) 32*, 1 (1985), 191–204.

[38] Elsheimy, F., Tsimos, G., and Papamanthou, C. Deterministic byzantine agreement with adaptive o(n· f) communication. *Symposium on Discrete Algorithms (SODA)* (2024), 1723.

[39] Fischer, M. J., Lynch, N. A., and Merritt, M. Easy Impossibility Proofs for Distributed Consensus Problems. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing, Minaki, Ontario, Canada, August 5-7, 1985* (1985), M. A. Malcolm and H. R. Strong, Eds., ACM, pp. 59–70.

[40] Galil, Z., Haber, S., and Yung, M. Cryptographic Computation: Secure Fault-Tolerant Protocols and the Public-Key Model. In *Conference on the Theory and Application of Cryptographic Techniques* (1987), Springer, pp. 135–155.

[41] Garg, S., Jain, A., Mukherjee, P., Sinha, R., Wang, M., and Zhang, Y. hints: Threshold signatures with silent setup. *IACR Cryptol. ePrint Arch.* (2023), 567.

[42] Gelles, Y., and Komargodski, I. Brief Announcement: Scalable Agreement Protocols with Optimal Optimistic Efficiency. *37th International Symposium on Distributed Computing, (DISC)* (2023), 42:1—-42:6.

[43] Gelles, Y., and Komargodski, I. Optimal Load-Balanced Scalable Distributed Agreement. In *IACR Cryptol. ePrint Arch.* (2023).

[44] Gilad, Y., Hemo, R., Micali, S., Vlachos, G., and Zeldovich, N. Algorand: Scaling Byzantine Agreements for Cryptocurrencies. In *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017* (2017), ACM, pp. 51–68.

[45] Gilbert, S., Lynch, N. A., and Shvartsman, A. A. Rambo: A Robust, Reconfigurable Atomic Memory Service for Dynamic Networks. *Distributed Comput. 23*, 4 (2010), 225–272.

[46] Groth, J. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II* (2016), M. Fischlin and J. Coron, Eds., vol. 9666 of *Lecture Notes in Computer Science*, Springer, pp. 305–326.

[47] Guerraoui, R., and Schiper, A. The Generic Consensus Service. *IEEE Trans. Software Eng. 27*, 1 (2001), 29–41.

[48] Haeberlen, A., Kouznetsov, P., and Druschel, P. Peerreview: practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007* (2007), T. C. Bressoud and M. F. Kaashoek, Eds., ACM, pp. 175–188.

[49] Helminger, L., Kales, D., Ramacher, S., and Walch, R. Multi-party revocation in sovrin: Performance through distributed trust. In *Topics in Cryptology - CT-RSA 2021 - Cryptographers' Track at the RSA Conference 2021, Virtual Event, May 17-20, 2021, Proceedings* (2021), K. G. Paterson, Ed., vol. 12704 of *Lecture Notes in Computer Science*, Springer, pp. 527–551.

[50] Kate, A., Zaverucha, G. M., and Goldberg, I. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings* (2010), M. Abe, Ed., vol. 6477 of *Lecture Notes in Computer Science*, Springer, pp. 177–194.

[51] King, V., Lonargan, S., Saia, J., and Trehan, A. Load balanced scalable byzantine agreement through quorum building, with full information. *Distributed Computing and Networking - 12th International Conference (ICDCN) 6522 LNCS* (2011), 203–214.

[52] King, V., and Saia, J. From almost everywhere to everywhere: Byzantine agreement with Õ (n3/2) bits. In *Distributed Computing, 23rd International Symposium (DISC)* (2009), vol. 5805 LNCS, pp. 464–478.

[53] King, V., and Saia, J. Breaking the $O(n^2)$ bit barrier: Scalable byzantine agreement with an adaptive adversary. *Journal of the ACM 58*, 4 (2011), 1–24.

[54] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. L. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007* (2007), T. C. Bressoud and M. F. Kaashoek, Eds., ACM, pp. 45–58.

[55] Lee, K. Decentralized threshold signatures for blockchains with non-interactive and transparent setup. *IACR Cryptol. ePrint Arch.* (2023), 1206.

[56] Lewis-Pye, A. Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model. *arXiv preprint arXiv:2201.01107* (2022).

[57] Lewis-Pye, A., and Abraham, I. Fever: Optimal Responsive View Synchronisation. *CoRR abs/2301.09881* (2023).

[58] Libert, B., Joye, M., and Yung, M. Born and Raised Distributively: Fully Distributed Non-Interactive Adaptively-Secure Threshold Signatures with Short Shares. *Theoretical Computer Science 645* (2016), 1–24.

[59] Lu, Y., Lu, Z., Tang, Q., and Wang, G. Dumbo-mvba: Optimal multi-valued validated asynchronous byzantine agreement, revisited. In *PODC '20: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, August 3-7, 2020* (2020), Y. Emek and C. Cachin, Eds., ACM, pp. 129–138.

[60] Momose, A., and Ren, L. Multi-Threshold Byzantine Fault Tolerance. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021* (2021), Y. Kim, J. Kim, G. Vigna, and E. Shi, Eds., ACM, pp. 1686–1699.

[61] Momose, A., and Ren, L. Optimal Communication Complexity of Authenticated Byzantine Agreement. In *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)* (2021), S. Gilbert, Ed., vol. 209 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 32:1–32:16.

[62] Nayak, K., Ren, L., Shi, E., Vaidya, N. H., and Xiang, Z. Improved extension protocols for byzantine broadcast and agreement. In *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference* (2020), H. Attiya, Ed., vol. 179 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 28:1–28:17.

[63] Nguyen, L. Accumulators from bilinear pairings and applications. In *Topics in Cryptology - CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14-18, 2005, Proceedings* (2005), A. Menezes, Ed., vol. 3376 of *Lecture Notes in Computer Science*, Springer, pp. 275–292.

[64] Pease, M. C., Shostak, R. E., and Lamport, L. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (1980), 228–234.

[65] Qiu, T., and Tang, Q. Predicate aggregate signatures and applications. *IACR Cryptol. ePrint Arch.* (2023), 1694.

[66] Rambaud, M. Bootstrapping Message-Linear-Constant-Round Consensus from a Bare PKI Setup , and Separation Bounds from the Idealized Message-Authentication Model. https://perso.telecom-paristech.fr/rambaud/articles/lower.pdf, 2023.

[67] Rambaud, M., Tonkikh, A., and Abspoel, M. Linear View Change in Optimistically Fast BFT. In *Proceedings of the 2022 ACM Workshop on Developments in Consensus* (2022), pp. 67–78.

[68] Reed, I. S., and Solomon, G. Polynomial odes over certain finite fields. *Journal of the society for industrial and applied mathematics 8*, 2 (1960), 300–304.

[69] Sheng, P., Wang, G., Nayak, K., Kannan, S., and Viswanath, P. Player-replaceability and forensic support are two sides of the same (crypto) coin. *IACR Cryptol. ePrint Arch.* (2022), 1513.

[70] Shoup, V. Practical threshold signatures. In *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding* (2000), B. Preneel, Ed., vol. 1807 of *Lecture Notes in Computer Science*, Springer, pp. 207–220.

[71] Spiegelman, A. In search for an optimal authenticated byzantine agreement. In *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)* (2021), S. Gilbert, Ed., vol. 209 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 38:1–38:19.

[72] Tsimos, G., Loss, J., and Papamanthou, C. Gossiping for Communication-Efficient Broadcast. *Advances in Cryptology - CRYPTO - 42nd Annual International Cryptology Conference 13509 LNCS* (2022), 439–469.

[73] Upfal, E. Tolerating linear number of faults in networks of bounded degree. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing* (New York, NY, USA, 1992), PODC '92, Association for Computing Machinery, p. 83–89.

[74] Wan, J., Momose, A., Ren, L., Shi, E., and Xiang, Z. On the Amortized Communication Complexity of Byzantine Broadcast. In *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing* (2023), pp. 253–261.

[75] Yin, M., Malkhi, D., Reiter, M. K., Golan-Gueta, G., and Abraham, I. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (2019), pp. 347–356.

A CRYPTOGRAPHIC SCHEMES

This section provides the formal definition of cryptographic accumulator and digest primitives.

Cryptographic accumulators. We follow the presentation of [62]. Let κ denote a security parameter, and let $D = \{d_1, d_2, \dots, d_n\}$ be a set of n values. An accumulator scheme consists of the following four elements:

- $ak \leftarrow \mathsf{Gen}(1^\kappa, n)$: An algorithm accepting a security parameter κ in unary notation 1^κ and a limit n on the set size, returns an accumulator key ak[12].
- $z \leftarrow \mathsf{Eval}(ak, D)$: An algorithm that takes the accumulator key ak and a set D of values, returns an accumulation value z for D.
- $w_i \leftarrow \mathsf{CreateWit}(ak, z, d_i)$: Given an accumulator key ak, an accumulation value z, and a value $d_i \in D$, this algorithm returns a witness w_i.

[12]Classic Bilinear Accumulator [63] requires Strong Diffie-Hellman (q-SDH) public parameters that can be costly to share in a secure manner for $t > n/3$ [29, 35]. Some variants allows a more practical distributed key generation (Gen) protocol [49].

- $\mathit{true}/\mathit{false} \leftarrow \mathsf{Verify}(ak, z, w_i, d_i)$: This algorithm takes an accumulator key ak, an accumulation value z, a witness w_i, and a value d_i. It returns true if w_i correctly corresponds to $d_i \in D$, and false otherwise, except with negligible probability $negl(\kappa)$.

This description deliberately leaves out auxiliary information *aux* typically included in cryptographic accumulators, as the bilinear accumulator used in our approach does not require *aux*. We assume Eval to be deterministic, which is the case with the bilinear accumulator we employ in ADA-DARE. Moreover, the accumulator scheme is assumed to be collision-free, i.e., for any accumulator key $ak \leftarrow \mathsf{Gen}(1^\kappa, n)$, it is computationally impossible to establish $(\{d_1, ..., d_n\}, d', w')$ such that (1) $d' \notin \{d_1, ..., d_n\}$, (2) $z \leftarrow \mathsf{Eval}(ak, \{d_1, ..., d_n\})$, and (3) $\mathsf{Verify}(ak, z, w', d') = \mathit{true}$.

Digests. Concretely, $\mathsf{Digest}(v) = \mathsf{Eval}(ak, \{(1, P_v(1)), ..., (n, P_v(n))\})$, where Eval is the accumulator evaluation function (see above), ak is the accumulator key (see above), and $[P_v(1), ..., P_v(n)] = \mathsf{encode}(v)$ is the Reed-Solomon encoding of value v (see Appendix C.1). The collision-resistance of the Digest function reduces to the collision-resistance of the underlying cryptographic accumulator scheme. We note that this construction is standard (see, e.g., [62]).

Multiverse threshold signatures. In ADA-DARE, we rely on a modern implementation [55, 65] of a multiverse threshold signature scheme (MTSS) [6, 41]. In such a scheme, each process P_i holds a distinct secret key sk_i; there exists a single public key pk_i. The multiverse threshold signature scheme (MTSS) is defined by the following primitives (with some potential public parameters obtained by a transparent setup):

- $(\mathrm{CK}_U, \mathrm{VK}_U) \leftarrow \mathsf{UniverseSetup}(U \subseteq \Pi)$: Given a *universe* (subset of processes) $U \subseteq \Pi$, the algorithm computes a combine key CK_U and a verification key VK_U. Importantly, no communication will be done by this algorithm.
- $\sigma \leftarrow \mathsf{Sign}(\mathrm{sk}, \mathrm{msg})$: Upon receiving a secret key sk and a message msg, produces a partial signature σ.
- $\mathit{true}/\mathit{false} \leftarrow \mathsf{PartialVerify}(\mathrm{msg}, \sigma, \mathrm{pk})$: This algorithm takes a message msg, a partial signature σ, and a public key pk, verifies the partial signature.
- $\sigma \leftarrow \mathsf{Combine}(\mathrm{CK}_U, \{\sigma_i\}_{i \in S \subseteq U})$: Given the combine key CK_U, and a set of signatures $\{\sigma_i\}_{i \in S \subseteq U}$, the algorithm outputs a succinct signature σ.
- $\mathit{true}/\mathit{false} \leftarrow \mathsf{Verify}(\mathrm{msg}, \sigma, T, \mathrm{VK}_U)$: This algorithm, upon receiving a message msg, a signature σ, a (dynamic) threshold T, and the verification key VK_U, verifies the signature iff it is signed by T different processes.

Similarly to the threshold signature schemes, each signature produced by the MTSS [55] has $O(\kappa)$. bits We define MTS partial signatures as the partial signatures produced under this scheme, and a (T, U)-MTS as a succinct signature under this scheme over a universe U and threshold T.

The Multiverse Threshold Signature scheme ensures the following:

- *Correctness*: Let $U \subseteq \Pi$ be a universe, $(\mathrm{CK}_U, \mathrm{VK}_U) \leftarrow \mathsf{UniverseSetup}(U)$, and m be a message. If $\sigma \leftarrow \mathsf{Combine}(\mathrm{CK}_U, \{\sigma_i\}_{i \in S \subseteq U})$, with $|S| = T$, where for each $i \in S$, $\sigma_i \leftarrow \mathsf{Sign}(\mathrm{sk}_i, m)$, then $\mathsf{Verify}(m, \sigma, T, \mathrm{VK}_U)$ returns *true*.
  Intuitively, if T valid signatures under different public keys from processes belonging to a certain universe U are used to produce the signature σ on some message m with combine , then the verification referring to the message m, the threshold T, the signature σ, and the universe U, returns *true*.
- *Unforgeability*: Let $U \subseteq \Pi$ be a universe, $(\mathrm{CK}_U, \mathrm{VK}_U) \leftarrow \mathsf{UniverseSetup}(U)$, m be a message, T be an integer, and σ be a signature. If $\mathsf{Verify}(m, \sigma, T, \mathrm{VK}_U)$ returns *true*, then $\sigma_i \leftarrow \mathsf{Sign}(\mathrm{sk}_i, m)$ has been computed for T different process $p_i \in U$.

Intuitively, the adversary cannot generate a valid (T, U)-MTS signature for some message m if it does not have T signatures from different signers on m, or the signer sets is not a subset of the corresponding universe U.

B ADA-DARE

This section fixes some generic validity val, and presents ADA-DARE$_{\text{val}}$ (ADAptively Disperse, Agree, REtrieve), which is composed of three algorithms:

(1) ADA-DISP$_{\text{val}}$, which disperses the proposals, valid for the specific validity property val;
(2) **CKS** [32], which ensures agreement on the digest of a previously dispersed proposal; and
(3) ADA-RETRIEVE, which rebuilds the proposal corresponding to the agreed-upon digest.

B.1 Building Blocks: Overview

In this subsection, we define the three building blocks of ADA-DARE$_{\text{val}}$. Concretely, we define their interface and properties, as well as their complexity.

B.1.1 ADA-DISP$_{\text{val}}$.

Interface & properties. In a nutshell, correct processes aim to collectively disperse some valid value $v \in \text{val}(c)$, for the specific validity property val and the input configuration[13] c: eventually, all correct processes acquire a digest along with a proof of dispersal (PoD), a threshold signature that proves the pre-image value of the digest is valid and has been successfully dispersed.

Concretely, ADA-DISP$_{\text{val}}$ exposes the following interface:

- **request** propose$(v \in \text{Value})$: a process proposes a value v; each correct process invokes propose(v) exactly with externallyValid$(v) = \textit{true}$.
- **indication** acquire$(d \in \text{Digest_Value}, \sigma^d \in \text{T_Signature})$: a process acquires a pair (d, σ^d).

We say that a correct process *obtains* a threshold signature (resp., a value) if and only if it stores the signature (resp., the value) in its local memory. (Obtained values can later be retrieved by all correct processes using ADA-RETRIEVE; see Appendix B.1.3 and Algorithm 3.) ADA-DISP$_{\text{val}}$ ensures the following:

- *Integrity:* If a correct process triggers acquire(d, σ^d), then CombinedVerify$^{t+1}(\langle \text{ACK}, d\rangle, \sigma^d)$) $= \textit{true}$.
- *Termination:* Every correct process eventually acquires at least one digest-signature pair.
- *Redundancy:* Let a correct process obtain (d, σ^d) such that CombinedVerify$^{t+1}(\langle \text{ACK}, d\rangle, \sigma^d)$ $= \textit{true}$, for some digest d and some threshold signature σ^d. Then, (at least) $t + 1 - f$ correct processes have obtained a value v such that (1) Digest$(v) = d$, and (2) $v \in \text{val}(c)$ for the input configuration c defined by the correct proposals. Let us remark that (2) might imply externallyValid$(v) = \textit{true}$ depending on val.

Note that it is not required for all correct processes to acquire the same digest value (nor the same threshold signature). Moreover, the specification allows for multiple acquired pairs.

Complexity. Both ADA-DISP$_{\text{val}_{ic}}$ and ADA-DISP$_{\text{val}_{sv}}$ exchange $O(nL_o + n(f + 1)\kappa)$ bits, where val$_{ic}$ and val$_{sv}$ stand for the validity properties of IC and SMVBA, respectively. Also, every correct process acquires at least one digest-signature pair by $O(f)$ rounds. Finally, all correct processes halt simultaneously within $O(n)$ rounds.

Implementation. The details on ADA-DISP's implementation are relegated to §6.

*B.1.2 **CKS**.*

Interface & properties. CKS is a VBA algorithm for some generic valid$_{\textbf{CKS}}(\cdot)$ predicate.[14]

[13]See Appendix C.1 for the definition of input configuration.

[14]Recall that the interface and properties of Byzantine consensus algorithms are introduced in §1.

In **CKS**, processes propose and decide pairs ($d \in \mathsf{Digest\_Value}, \sigma_d \in \mathsf{T\_Signature}$); moreover, $\mathsf{valid}_{\mathbf{CKS}}(d, \sigma^d) \equiv \mathsf{CombinedVerify}^{t+1}(\langle \textsc{ack}, d \rangle, \sigma^d)$. In other words, a valid value must be in the form of a digest accompanied by its PoD.

Complexity. **CKS** achieves $O(\kappa n(f+1))$ bit complexity for $O(\kappa)$-bits values and correct processes decide in $O(n)$ rounds.

Implementation. In a nutshell, **CKS** is a validated BA protocol that follows the silent views paradigm. Under the hood, it employs a fallback protocol in a scenario when there are too many faulty processes and thus, unable to decide in its optimistic path. Turning the fallback protocol [61] into a validated BA protocol is straightforward, by ignoring each received message containing a non externally-valid value (details are given in Appendix D). Moreover, **CKS** incorporates the correction of a minor technical flaw in [32] addressed by [38].

B.1.3 ada-Retrieve.

Interface & properties. In ada-Retrieve, each correct process starts with (1) a digest and (2) either (a) some corresponding pre-image value, or (b) $\bot$. Eventually, all correct processes output the same value (the pre-image). Formally, ada-Retrieve exposes the following interface:

- **request** $\mathsf{input}(d \in \mathsf{Digest\_Value}, v \in \mathsf{Value} \cup \{\bot\})$: a process inputs a digest d and either $\bot$ or a value v such that $\mathsf{Digest}(v) = d$; each correct process invokes $\mathsf{input}(\cdot)$ exactly once. Moreover, the following is assumed:
  - No two correct processes invoke $\mathsf{input}(d_1, v_1)$ and $\mathsf{input}(d_2, v_2)$ with $d_1 \neq d_2$.
  - At least $t+1-f$ correct processes invoke $\mathsf{input}(d, v)$ with $v \neq \bot$ (i.e., $\mathsf{Digest}(v) = d$).
- **indication** $\mathsf{output}(v' \in \mathsf{Value})$: a process outputs a value v'.

The following properties are ensured:

- *Agreement:* No two correct processes output different values.
- *Validity:* Let a correct process input a value v. No correct process outputs a value $v' \neq v$.
- *Termination:* Every correct process eventually outputs a value.

Complexity. ada-Retrieve exchanges $O(nL_o + n(f+1)\kappa)$ bits. Moreover, ada-Retrieve terminates in $O(\log n)$ rounds.

Implementation. The details on ada-Retrieve's implementation are relegated to §5.

B.2 Pseudocode

Algorithm 3 gives ada-Dare$_{\mathsf{val}}$'s pseudocode. We explain it from the perspective of a correct process p_i. An execution of ada-Dare$_{\mathsf{val}}$ consists of three phases (each of which corresponds to one building block).

During the *dispersal* phase, p_i executes ada-Disp$_{\mathsf{val}}$ using its proposal v_i (line 6). Eventually, p_i acquires a digest-signature pair (d_i, σ^{d_i}) (line 7) due to the termination property of ada-Disp$_{\mathsf{val}}$. Moreover, $\mathsf{valid}_{\mathbf{CKS}}(d_i, \sigma^{d_i})$ holds due to the validity property of ada-Disp$_{\mathsf{val}}$.

Next, in the *agreement* phase, p_i proposes the previously acquired digest-signature pair (d_i, σ^{d_i}) to **CKS** (line 9). As **CKS** satisfies termination, agreement, and external validity, all correct processes eventually agree on a digest-signature pair (d, σ^d) (line 10).

Finally, in the *retrieval* phase, once p_i decides (d, σ^d) from **CKS**, it checks whether it has previously obtained a value v with $\mathsf{Digest}(v) = d$ (line 11). If it has, p_i inputs (d, v) to ada-Retrieve; otherwise, p_i inputs $(d, \bot)$ (line 13). Observe that here, the pre-conditions of ada-Retrieve are met: all correct processes input the same digest (due to the agreement property of **CKS**) and at least $t+1-f$ correct processes input a value $v \neq \bot$ with $\mathsf{Digest}(v) = d$ (due to the redundancy property of ada-Dare$_{\mathsf{val}}$). Therefore, all correct processes (including p_i) eventually output the same value v from ada-Retrieve (due to the termination and validity properties of ada-Retrieve; line 14),

Algorithm 3 ADA-DARE$_{\mathsf{val}}$: Pseudocode (for process p_i)

```
1: Uses:
2:     ADA-DISP_val, instance disperser            ▷ bits: $O(nL_o + n(f+1)\kappa)$, latency: $O(n)$ (see §6)
3:     CKS, instance agreement                     ▷ bits: $O(\kappa n(f+1))$, latency: $O(n)$ (see Appendix D)
4:     ADA-RETRIEVE, instance retriever            ▷ bits: $O(nL_o + n(f+1)\kappa)$, latency: $O(\log n)$ (see §5)
5: upon propose($v_i \in$ Value):
6:     invoke disperser.propose($v_i$)
7: upon disperser.acquire($d_i \in$ Digest_Value, $\sigma^{d_i} \in$ T_Signature):
8:     wait until $O(n)$ rounds has passed since invoking disperser ▷ so all processes invoke agreement simultaneously
9:     invoke agreement.propose($d_i, \sigma^{d_i}$)
10: upon agreement.decide($d \in$ Digest_Value, $\sigma^d \in$ T_Signature):
11:     $v \leftarrow$ an obtained value such that Digest($v$) $= d$ (if such a value was not obtained, $v = \bot$)
12:     wait until $O(n)$ rounds has passed since invoking agreement ▷ so all processes invoke retriever simultaneously
13:     invoke retriever.input($d, v$)
14: upon retriever.output(Value $v'$):
15:     trigger decide($v'$)
```

which represents the decision of ADA-DARE$_{\mathsf{val}}$ (line 15). Note that v is valid according to val due to the redundancy of ADA-DISP$_{\mathsf{val}}$ and the external validity of **CKS**.

B.3 Analysis

We start by proving the correctness of ADA-DARE$_{\mathsf{val}}$.

THEOREM 1. *ADA-DARE$_{\mathsf{val}}$ is correct.*

PROOF. Every correct process starts the dispersal phase with its proposal (line 6). Due to the termination and integrity property of ADA-DISP$_{\mathsf{val}}$, every correct process eventually acquires a digest-signature pair (line 7). Hence, every correct process eventually proposes to **CKS** (line 9), which implies that every correct process eventually decides the same digest-signature pair (d, σ^d) from **CKS** (line 10) due to the agreement and termination properties of **CKS**. As (d, σ^d) is decided by all correct processes, at least $t + 1 - f$ correct processes p_i have obtained a value v such that (1) $\mathsf{Digest}(v) = d$, and (2) $v \in \mathsf{val}(c)$ (due to the redundancy property of ADA-DISP$_{\mathsf{val}}$ and $\mathsf{valid}_{\mathbf{CKS}}(\cdot)$ predicate of **CKS**). Therefore, all of these correct processes input v to ADA-RETRIEVE (line 13). Moreover, no correct process inputs a different digest. Thus, the conditions required by ADA-RETRIEVE are met, which implies that all correct processes eventually output the same valid value (namely, v) from ADA-RETRIEVE (line 14), and decide it (line 15). □

Next, we prove the complexity of ADA-DARE$_{\mathsf{val}}$.

THEOREM 2. *ADA-DARE$_{\mathsf{val}}$ achieves $O(nL_o + n(f+1)\kappa)$ bit complexity and $O(n)$ latency if* $\mathsf{val} \in \{\mathsf{val}_{ic}, \mathsf{val}_{sv}\}$.

PROOF. As ADA-DARE$_{\mathsf{val}}$ is a sequential composition of its building blocks, its bit complexity (resp. latency) is the sum of the bit complexity (resp. latency) of (1) ADA-DISP$_{\mathsf{val}}$, (2) **CKS**, and (3) ADA-RETRIEVE. □

C ADA-DISP

This section proves that ADA-DISP solves the dispersion problem (defined in Appendix B.1.1). Moreover, this section shows that ADA-DISP$_{\mathsf{val}_{ic}}$ and ADA-DISP$_{\mathsf{val}_{sv}}$, two specific instances of ADA-DISP, achieve optimal word complexity of $O(nL_o + n(f+1)\kappa)$ while solving the dispersion problem related to interactive consistency and strong & external validity, respectively.

C.1 Preliminaries

Specific validity property of ADA-DARE. To define a specific Byzantine agreement problem ADA-DARE solves, we need a generic definition of the validity property. To this end, we reuse existing formalism and nomenclature [24, 26] revolving around the notion of input configurations. Roughly speaking, an input configuration maps correct processes to their proposals, while a validity property maps an input configuration into a set of admissible decisions.

Let a *process-proposal* pair be a pair (p_i, v), where $p_i \in \Pi$ is a process and $v \in \mathsf{Value}_I$ is a proposal. Given any process-proposal pair $pp = (p_i, v)$, we denote by $\mathsf{proposal}(pp) = v$ the proposal associated with the pair. An *input configuration* is a tuple $\left[pp_1, pp_2, ..., pp_x\right]$ such that (1) $n - t \leq x \leq n$, and (2) every process-proposal pair is associated with a distinct process. In summary, an input configuration is an assignment of proposals to (all) correct processes.

The set of all input configurations is denoted by $\mathcal{I}$. Moreover, $\mathcal{I}_x \subsetneq \mathcal{I}$ denotes the set of all input configurations with exactly x process-proposals pairs. Given any input configuration $c \in \mathcal{I}$, $c[i]$ denotes the process-proposal pair associated with the process p_i; if such a process-proposal pair does not exist, $c[i] = \bot$. Moreover, $\pi(c) = \{p_i \in \Pi \mid c[i] \neq \bot\}$ denotes the set of all correct processes according to any input configuration $c \in \mathcal{I}$.

Let $\mathcal{E}$ be any execution of any distributed algorithm $\mathcal{A}$, and let $c \in \mathcal{I}$ be any input configuration. We say that $\mathcal{E}$ *corresponds to* c (in short, $\mathsf{input\_conf}(\mathcal{E}) = c$) if and only if:

- $Correct_{\mathcal{A}}(\mathcal{E}) = \pi(c)$; and
- for every process $p_i \in \pi(c)$, $\mathsf{input}(p_i) = \mathsf{proposal}(c[i])$.[15]

A validity property val is a function $\mathsf{val} : \mathcal{I} \to 2^{\mathsf{Value}_O}$ such that $\mathsf{val}(c) \neq \emptyset$, for every input configuration $c \in \mathcal{I}$. Finally, we say that ADA-DARE *satisfies* a validity property val if and only if, in any execution $\mathcal{E} \in execs(\text{ADA-DARE})$, no correct process decides any value $v' \notin \mathsf{val}\big(\mathsf{input\_conf}(\mathcal{E})\big)$[16]. Intuitively, ADA-DARE satisfies a validity property if correct processes only decide admissible values.

Using the introduced formalism, interactive consistency can be defined as

$$\mathsf{val}_{ic}(c) = \{c' \in \mathcal{I}_n \mid \forall p_i \in \pi(c), c[i] = c'[i]\},$$

while strong & external validity (for which $\mathsf{Value}_I = \mathsf{Value}_O$) can be defined as

$$\mathsf{val}_{sv}(c) = \begin{cases} v \in \mathsf{Value}_O, & \text{if } \exists v \in \mathsf{Value}_I : \forall p_i \in \pi(c), \mathsf{proposal}(c[i]) = v \\ \{v' \in \mathsf{Value}_O \mid \mathsf{externallyValid}(v')\}, & \text{otherwise.} \end{cases}$$

Entries. To efficiently satisfy different validity properties, processes collect the entries of other processes, where an entry can be either (1) a proposal, or (2) a digest of a proposal. Hence, for each validity property val, we define $\mathsf{Entry}_{\mathsf{val}} \in \{\mathsf{Value}, \mathsf{Digest\_Value}\}$. For example, $\mathsf{Entry}_{\mathsf{val}_{ic}} = \mathsf{Value}$, while $\mathsf{Entry}_{\mathsf{val}_{sv}} = \mathsf{Digest\_Value}$. We denote by $\mathsf{entry}_{\mathsf{val}} : \mathsf{Value}_I \to \mathsf{Entry}_{\mathsf{val}}$ the function that maps an input value to its corresponding entry depending on a specific validity property val. For example, $\mathsf{entry}_{\mathsf{val}}(v) = v$ given that $\mathsf{Entry}_{\mathsf{val}} = \mathsf{Value}$, and $\mathsf{entry}_{\mathsf{val}}(v) = \mathsf{Digest}(v)$ given that $\mathsf{Entry}_{\mathsf{val}} = \mathsf{Digest\_Value}$.

Verifiable entry vector. A key step of our concrete solution for the dispersion problem is achieving verifiable vector collection. To define it, we need to define what an entry vector is, and what we mean by "verifiable". Let val be any fixed validity property. An entry vector for val is a vector $vec = (e_1, ..., e_n) \in (\mathsf{Entry}_{\mathsf{val}} \cup \{\bot\})^n$. We denote by $\mathsf{EntryVector}_{\mathsf{val}}$ the set of entry vectors for validity property val.

[15]Observe that this definition does not apply only to ADA-DARE.

[16]To be more formal, we should require that for every adversary Adv, for every input configuration c, for every round number r, for every security number λ, $\mathbb{P}\mathrm{rob}_{(\mathcal{A},Adv,c,r,\lambda)}(\{\mathcal{E}_{sad} \in \mathsf{Execs}_{(\mathcal{A},Adv,c,r,\lambda)} | \mathcal{E}_{sad} \text{ violates val}\}) = 1 - neg(\lambda)$.

A vector predicate for val is a mapping $\mathsf{EntryVector}_{\mathsf{val}} \times \{0,1\}^* \to \{true, false\}$. A vector proof *vec_proof* for an entry vector $vec = (e_1, ..., e_n)$, and a vector predicate vec_pred, is a string satisfying $\text{vec\_pred}(vec, vec\_proof) = true$. When the vector predicate is clear in the context, we say *vec_proof* is a vector proof for *vec*.

Let vec_pred be a vector predicate for val, and let $\mathcal{A}$ be any distributed algorithm. We say that vec_pred is *unforgeable* for $(\text{val}, \mathcal{A})$, if and only if, for every execution $\mathcal{E} \in execs(\mathcal{A})$ corresponding to some input configuration $c \in \mathcal{I}$, it is computationally infeasible to forge a pair $((e'_1, ..., e'_n) \in \mathsf{EntryVector}_{\mathsf{val}}, proof \in \{0,1\}^*)$ that satisfies the following two conditions: (1) $\text{vec\_pred}\big((e'_1, ..., e'_n), proof\big) = true$, and (2) $e'_j \neq \mathsf{entry}_{\mathsf{val}}\big(\mathsf{proposal}(c[j])\big)$, for some correct process $p_j \in \mathsf{Correct}_{\mathcal{A}}(\mathcal{E})$.

Accuracy and vector verification predicate in our concrete implementation. The verifiability principle mentioned above will be realized via ACCUSE and DISCLOSE messages. Intuitively, an $\langle \text{ACCUSE}, p_a, \sigma_i \rangle$ message with $\mathsf{ShareVerify}_i^{t+1}(\langle \text{ACCUSE}, p_a \rangle, \sigma_i) = true$ means "I, p_i, claim that p_a is faulty", while a $\langle \text{DISCLOSE}, e_i, \sigma_i \rangle$ message, with $\mathsf{ShareVerify}_i^{t+1}(\langle \text{DISCLOSE}, e_i \rangle, \sigma_i) = true$, means "I, p_i, declare that my entry is e_i". These two categories of messages will be enough to define our security properties.

Next, we introduce the definition of local accuracy.[17]

Definition 1 (Local accuracy). Let val be any validity property. We say that any distributed algorithm $\mathcal{A}$ satisfies val-*local-accuracy* if, for every execution $\mathcal{E} \in execs(\mathcal{A})$ corresponding to some input configuration $c \in \mathcal{I}$, for every pair of correct processes $(p_i, p_j) \in (Correct_{\mathcal{A}}(\mathcal{E}))^2$:

(1) p_i never partially-signs an $\langle \text{ACCUSE}, p_j \rangle$ message.
(2) p_i never partially-signs a $\langle \text{DISCLOSE}, e'_i \rangle$ message for $e'_i \neq \mathsf{entry}_{\mathsf{val}}(\mathsf{proposal}(c[i]))$.

Intuitively, local accuracy ensures that no correct process is ever exposed by a correct process, or provably associated with an entry different from its own. We next introduce the concept of global accuracy.

Definition 2 (Global accuracy). Let val be any validity property. We say that any distributed algorithm $\mathcal{A}$ satisfies val-*global-accuracy* if, for every execution $\mathcal{E} \in execs(\mathcal{A})$ corresponding to some input configuration $c \in \mathcal{I}$, for every correct process $p_j \in Correct_{\mathcal{A}}(\mathcal{E})$, it is computationally infeasible to obtain a signature σ such that either:

(1) $\mathsf{CombinedVerify}^{t+1}(\langle \text{ACCUSE}, p_j \rangle, \sigma) = true$, or
(2) $\mathsf{ShareVerify}_j^{t+1}(\langle \text{DISCLOSE}, e_j \rangle, \sigma) = true$, for $e'_j \neq \mathsf{entry}_{\mathsf{val}}(\mathsf{proposal}(c[j]))$.

The following lemma proves that local accuracy implies global accuracy.

Lemma 6 (From local accuracy to global accuracy). Let val be any validity property. Let $\mathcal{A}$ be any distributed algorithm such that $\mathcal{A}$ satisfies val-local-accuracy. Then, $\mathcal{A}$ satisfies val-global-accuracy.

PROOF. By contradiction, suppose that $\mathcal{A}$ does not satisfy val-global-accuracy. Hence, a signature σ is obtained such that (1) $\mathsf{CombinedVerify}^{t+1}\big(\langle \text{ACCUSE}, p_j \rangle, \sigma\big) = true$, where p_j is any correct process, or (2) $\mathsf{ShareVerify}_j^{t+1}\big(\langle \text{DISCLOSE}, e_j \rangle, \sigma\big) = true$, for $e_j \neq \mathsf{entry}_{\mathsf{val}}(\mathsf{proposal}(c[j]))$. However, this is impossible given that $\mathcal{A}$ satisfies val-local-accuracy. □

Moreover, we define the specific vector predicate of the algorithm ADA-DISP.

Definition 3 (Vector predicate $\mathsf{VerifyVector}_{\mathsf{val}}$). Let val be any validity property. $\mathsf{VerifyVector}_{\mathsf{val}}$ is defined as a vector predicate for val such that, for every entry vector $vec = (e_1, ..., e_n) \in$

[17]The name comes from the ideal accuracy property of accountable protocols that states that no correct process is ever exposed by a correct process [48].

$\mathsf{EntryVector}_{\mathsf{val}}$ for val, for every auxiliary string $aux \in \{0,1\}^*$, $\mathsf{VerifyVector}_{\mathsf{val}}(vec, aux)$ returns *true* if and only if:

- aux is of the form $(\sigma_1, ..., \sigma_n) \in (\{0,1\}^\kappa)^n$ such that, for every $j \in [1:n]$:
  - if $e_j = \bot$, then $\mathsf{CombinedVerify}^{t+1}(\langle \textsc{accuse}, p_j \rangle, \sigma_j) = true$
  - if $e_j \neq \bot$, then $\mathsf{ShareVerify}_j^{t+1}(\langle \textsc{disclose}, e_j \rangle, \sigma_j) = true$

Intuitively, a proof is well-formed if for each process p_j associated to some entry e_j, if e_j is a $\bot$-value, then it is backed by an undeniable proof-of-misbehaviour against p_j ($\mathsf{PoM}(p_j)$), and otherwise, e_j is backed by a corresponding DISCLOSE message partially-signed by process p_j. We can see the obvious connection between accuracy and unforgeability of $\mathsf{VerifyVector}_{\mathsf{val}}$.

Lemma 7 (From global accuracy to $\mathsf{VerifyVector}_{\mathsf{val}}$'s unforgeability)**.** Let val be any validity property. If ADA-DISP$_{\mathsf{val}}$ satisfies val-global-accuracy, then $\mathsf{VerifyVector}_{\mathsf{val}}$ is unforgeable for (val, ADA-DISP$_{\mathsf{val}}$).

PROOF. Let val be any validity property. Let $\mathcal{E} \in execs(\text{ADA-DISP}_{\mathsf{val}})$. Assume the computation of $(vec = (e'_1, ..., e'_n) \in \mathsf{EntryVector}_{\mathsf{val}}, vec\_proof = (\sigma_1, ..., \sigma_n) \in (\{0,1\}^\kappa)^n)$ in $\mathcal{E}$, such that (i) $\mathsf{VerifyVector}_{\mathsf{val}}\ (vec, vec\_proof) = true$ and (ii) $e'_j \neq \mathsf{entry}_{\mathsf{val}}(\mathsf{proposal}(\mathsf{input\_conf}(\mathcal{E})[j]))$ for some correct process $p_j \in \mathsf{Correct}_{\mathcal{A}}(\mathcal{E})$.

There are 2 cases: (1) $e'_j = \bot$ and $\mathsf{CombinedVerify}^{t+1}(\langle \textsc{accuse}, p_j \rangle, \sigma_j)$, and (2) $e'_j \neq \bot$ and $\mathsf{ShareVerify}^{t+1}(\langle \textsc{disclose}, e'_j \rangle, \sigma_j)$. However, both are impossible as ADA-DISP$_{\mathsf{val}}$ satisfies val-global-accuracy. □

It will be easy to show that all the components of ADA-DISP, and thus ADA-DISP itself, satisfy accuracy. For presentation's sake, we assume the val_{ic}-local-accuracy and the val_{sv}-local-accuracy of ADA-DISP$_{\mathsf{val}_{ic}}$ and ADA-DISP$_{\mathsf{val}_{sv}}$, respectively. The result will be proven in the later stages of the appendix.

Lemma 8 (ADA-DISP's local accuracy)**.** The following holds:
- ADA-DISP$_{\mathsf{val}_{ic}}$ satisfies val_{ic}-local-accuracy.
- ADA-DISP$_{\mathsf{val}_{sv}}$ satisfies val_{sv}-local-accuracy.

PROOF. This lemma is proven later as Lemma 27. □

Consequently, we obtain the following two intermediate results.

Lemma 9 (ADA-DISP's global accuracy)**.** The following holds:
- ADA-DISP$_{\mathsf{val}_{ic}}$ satisfies val_{ic}-global-accuracy.
- ADA-DISP$_{\mathsf{val}_{sv}}$ satisfies val_{sv}-global-accuracy.

PROOF. Follows immediately from Lemma 8 and Lemma 6. □

Lemma 10 ($\mathsf{VerifyVector}_{\mathsf{val}_{ic}}$'s and $\mathsf{VerifyVector}_{\mathsf{val}_{sv}}$'s unforgeability)**.** The following holds:
- $\mathsf{VerifyVector}_{\mathsf{val}_{ic}}$ is unforgeable for (ADA-DISP$_{\mathsf{val}_{ic}}$, val_{ic}).
- $\mathsf{VerifyVector}_{\mathsf{val}_{sv}}$ is unforgeable for (ADA-DISP$_{\mathsf{val}_{sv}}$, val_{sv}).

PROOF. Follows immediately from Lemma 9 and Lemma 7. □

C.2 The Dispersion Problem

In this subsection, we recall the specification of the dispersion problem (Appendix C.2.1), and give an overview of ADA-DISP, our implementation of the problem (Appendix C.2.2).

C.2.1 Specification. Dispersion is a problem, parameterized by a specific validity property val, in which every process starts with its input value and outputs a digest $d \in \{0,1\}^\kappa$ and an associated proof-of-dispersal $\mathsf{PoD}(d)$ proving that the corresponding pre-image value v for which $\mathsf{Digest}(v) = d$ is (1) valid according to val, and (2) is previously observed by at least $t+1-f$ correct processes. Concretely, the proof of dispersal is a $(t+1)$-threshold signature of an acknowledgement message confirming both validity and observation: $\mathsf{PoD}(d) = \mathsf{Combine}^{t+1}\big(\{\mathsf{ShareSign}_i^{t+1}(\langle \textsc{ack}, d\rangle)\}_{i\in S, |S|=t+1}\big)$.

Formally, the dispersion problem exposes the following interface:

- **request** $\mathsf{propose}(v \in \mathsf{Value})$: a process proposes a value v; a correct process invokes $\mathsf{propose}(v)$ only if $\mathsf{externallyValid}(v) = \mathit{true}$.
- **indication** $\mathsf{acquire}(d \in \mathsf{Digest\_Value}, \sigma^d \in \mathsf{T\_Signature})$: a process acquires a pair (d, σ^d).

We say that a correct process *obtains* a digest-signature pair (resp., a value) if and only if it stores the digest-signature pair (resp., the value) in its local memory. (Obtained values can later be retrieved by all correct processes using our retrieval algorithm ADA-RETRIEVE; see §4.3 and Algorithm 3.) The following properties are required by the dispersion problem:

- *Integrity:* If a correct process triggers $\mathsf{acquire}(d, \sigma^d)$, then $\mathsf{CombinedVerify}^{t+1}(\langle \textsc{ack}, d\rangle, \sigma^d)) = \mathit{true}$.
- *Termination:* Every correct process eventually acquires at least one digest-signature pair.
- *Redundancy:* Let a correct process obtain (d, σ^d) such that $\mathsf{CombinedVerify}^{t+1}(\langle \textsc{ack}, d\rangle, \sigma^d) = \mathit{true}$, for some digest d and some threshold signature σ^d. Then, there exists a pre-image v of d, satisfying $\mathsf{Digest}(v) = d$, such that the following holds:
  - Observation: (at least) $t+1-f$ correct processes have obtained v; and
  - Validity: $v \in \mathsf{val}(c)$, where c denotes the input configuration of the ongoing execution.

Note that it is not required for all correct processes to acquire the same digest value (nor the same threshold signature). Moreover, the specification allows for multiple pairs to be acquired by correct processes.

C.2.2 ADA-DISP's overview. ADA-DISP (Algorithm 5) operates across n views, each with (1) a different leader, and (2) $O(1)$ rounds. The satisfaction of the dispersion problem's properties is associated with the delivery of some proof-of-dispersal (PoD) by each correct process (to ensure the termination property). Each view of the ADA-DISP protocol is separated into two phases: (1) the *catch-up phase*, in which a correct leader disseminates an already-obtained PoD to processes that have not yet acquired their PoD, and (2) the *attempt phase*, in which a correct leader that has not previously obtained a PoD aims to enable all correct processes to acquire a PoD. Importantly, the attempt phase of a view with a correct leader uses communication (i.e., is not silent) *only if* no correct process has previously obtained a PoD.

Let us briefly explain the attempt phase of a view. The attempt phase consists of executing the protocol pod_creation_attempt (Algorithm 11), where the leader expects to collect a vector of entries of all potentially correct processes (i.e., processes that have not been previously provably detected). If the leader fails to collect the aforementioned vector, it accuses a new process p_a whose entry is missing, and encourages other processes to initiate the LearnOrExpose procedure (Algorithm 6) to ascertain p_a's input value or announce p_a faulty. If the leader succeeds in collecting the vector (and is correct), the verifiable vector collection (VVC) succeeds. In this case, the leader is able to construct and disseminate a PoD to all processes via the persuade protocol (see Algorithm 7 and Algorithm 8). An overview of the aforementioned "verifiable vector collection (VVC) + persuasion" structure of ADA-DISP is depicted in Figure 5 and Figure 6. The following subsections provide full details on all ADA-DISP's subprotocols (pod_creation_attempt, LearnOrExpose, persuade).

ADA-DISP's pseudocode. We explain the pseudocode of ADA-DISP (Algorithm 5) from the standpoint of a correct process p_i. Starting a certain view, process p_i starts the catch-up phase (Algorithm 4) at line 12; we fully explain the catch-up phase in the next paragraph. Then, process p_i engages in the pod_creation_attempt subprotocol (Algorithm 11) at line 16 if and only if p_i has not obtained a digest-PoD from the catch-up phase. In the final round, if p_i has previously initiated the pod_creation_attempt subprotocol and the leader of the view is correct, p_i either (1) successfully obtains a PoD that it retains (line 19), or (2) triggers a LearnOrExpose instance for a newly accused process p_a (we provide full details in Algorithm 11). If a new instance of LearnOrExpose is indeed initiated, then every correct process either gains knowledge of p_a's input or provably detects (i.e., exposes) p_a. Upon completion of all n views, p_i concludes its participation in ADA-DISP (line 22).

Let us now explain the catch-up protocol (Algorithm 4). In the first round, p_i checks whether it has already obtained a PoD (line 5). If it has not, p_i either (1) broadcasts an AID_REQ message (line 7) if it is the current leader, or (2) sends an AID_REQ message to the current view's leader (line 9). If p_i has already obtained a PoD and it receives an AID_REQ message (1) from the leader, or (2) being the leader itself (line 11), p_i transmits the already-obtained PoD to the sender of the AID_REQ message (line 12). In the third round, if p_i receives a PoD (line 14), it checks if it already has a PoD (line 15). If so, it moves on to the fourth round. If it does not, it obtains the freshly received PoD (line 16), and if p_i is the leader, it forwards the obtained PoD to every process (line 18). Finally, in the fourth round, if p_i receives a PoD from the leader (line 20), it obtains the received PoD (line 21).

Algorithm 4 catch_up$_i$(p_ℓ, *certified_digest*$_i$, *pod*$_i$): Pseudocode (for process p_i)

```
1: Input Parameters:
2:     Digest_Value certified_digest_i
3:     PoD pod_i
4: Round 1:
5:     if $pod_i = \bot$:
6:         if $p_i = p_\ell$:                                   ▷ if $p_\ell$ is the leader of this instance
7:             broadcast ⟨AID_REQ⟩
8:         else:
9:             send ⟨AID_REQ⟩ to $p_\ell$
10: Round 2:
11:     if $pod_i \neq \bot$, and ⟨AID_REQ⟩ is received from $p_j$, such that $p_\ell \in \{p_i, p_j\}$ :
12:         send ⟨AID_REPLY, certified_digest_i, pod_i⟩ to $p_j$
13: Round 3:
14:     if ⟨AID_REPLY, Digest_Value $d'$, PoD $\Sigma'$⟩ is received and $\text{CombinedVerify}^{t+1}(\langle \text{ACK}, d' \rangle, \Sigma') = true$:
15:         if $pod_i = \bot$:
16:             $certified\_digest_i \leftarrow d'$; $pod_i \leftarrow \Sigma'$
17:             if $p_i = p_\ell$:
18:                 broadcast ⟨AID_RELAY, certified_digest_i, pod_i⟩
19: Round 4:
20:     if ⟨AID_RELAY, Digest_Value $d'$, PoD $\Sigma'$⟩ is received from $p_\ell$ and $\text{CombinedVerify}^{t+1}(\langle \text{ACK}, d' \rangle, \Sigma') = true$:
21:         $certified\_digest_i \leftarrow d'$; $pod_i \leftarrow \Sigma'$
22:     return (certified_digest_i, pod_i)
```

Initial analysis of ADA-DISP. We now provide an initial analysis of ADA-DISP. First, we prove that, if (1) a correct process starts the catch_up protocol with a well-formed digest-signature pair and (2) the leader is correct, then every correct process returns a well-formed digest-signature pair at the end of the catch_up protocol.

Algorithm 5 ADA-DISP$_{\text{val}}$($start\_value_i$): Pseudocode (for process p_i)

```
1: Input Parameters:
2:     Value $start\_value_i \leftarrow v$, where $p_i$ has previously invoked start($v$)
3: Variables:
4:     Digest_Value $certified\_digest_i \leftarrow \bot$
5:     PoD $pod_i \leftarrow \bot$                                   ▷ succinct proof of dispersal
6:     Map(Process → Disclose_Msg) $indirects_i \leftarrow \emptyset$   ▷ values indirectly obtained through LearnOrExpose instances
7:     Map(Process → T_Signature) $culprits_i \leftarrow \emptyset$    ▷ PoM indirectly obtained through LearnOrExpose instances
8:     Set(Process) $contact_i^{in} \leftarrow \emptyset$    ▷ Processes that have requested the input of $p_i$ in some LearnOrExpose instance
9:     Set(Process) $contact_i^{out} \leftarrow \emptyset$   ▷ Processes whose input has been asked by $p_i$ in some LearnOrExpose instance
10: for $j = 1$ to $n$:
11:     Round 1:
12:         invoke catch_up($p_j$, $certified\_digest_i$, $pod_i$)
13:     Round 4:
14:         $certified\_digest_i$, $pod_i$ ← outputs of catch_up($p_j$, $certified\_digest_i$, $pod_i$)
15:         if $pod_i = \bot$:
16:             invoke pod_creation_attempt$_{\text{val}}$($start\_value_i$, $p_j$, $indirects_i$, $culprits_i$, $contact_i^{in}$, $contact_i^{out}$)
17:     Round 4+$r_{\text{val}}^{pca}$:          ▷ $r_{\text{val}}^{pca} \in O(1)$ is the round complexity of pod_creation_attempt$_{\text{val}}$
18:         if pod_creation_attempt$_{\text{val}}$ has been previously invoked:
19:             $certified\_digest_i$, $pod_i$, $indirects_i$, $culprits_i$, $contact_i^{in}$, $contact_i^{out}$ ← outputs of pod_creation_attempt$_{\text{val}}$
20:         if $pod_i \neq \bot$:
21:             trigger acquire($certified\_digest_i$, $pod_i$)
22: trigger stop
```

Lemma 11. Let (1) a correct process start the catch_up protocol (Algorithm 4) with a well-formed digest-signature pair, and (2) the leader be correct. Then, by the end of the execution of catch_up protocol, every correct process returns a well-formed digest-signature pair.

PROOF. Let p_ℓ, p_j be processes such that (1) p_ℓ is the correct leader, and (2) p_j has previously acquired a digest-signature pair $(d_j, \text{PoD}(d_j))$. We consider two cases:

- Suppose that p_ℓ has not previously obtained a digest-signature pair; note that $p_\ell \neq p_j$. Then, p_ℓ broadcasts an AID_REQ message (line 7). Upon the reception of this AID_REQ message (line 11), process p_j replies with an AID_REPLY message (line 12) holding its digest-signature pair $(d_j, \text{PoD}(d_j))$. Upon the reception of this AID_REPLY message (line 14), process p_ℓ acquires the digest-signature pair $(d_j, \text{PoD}(d_j))$ and broadcasts an AID_RELAY message (line 18) holding the digest-signature pair $(d_j, \text{PoD}(d_j))$. This message is received by every correct process (line 20), allowing the delivery (line 21) and return of $(d_j, \text{PoD}(d_j))$ (line 22).
- Suppose that p_ℓ has previously obtained a digest-signature pair; note that p_ℓ might be p_j. Then any correct process p_i without a digest-signature pair sends an AID_REQ message (line 9) to p_ℓ. Upon the reception of this AID_REQ message (line 11), process p_ℓ replies with an AID_REPLY message (line 12) holding its digest-signature pair $(d_j, \text{PoD}(d_j))$. Upon the reception of this AID_REPLY message (line 14), process p_i delivers and returns (line 22) the digest-signature pair $(d_j, \text{PoD}(d_j))$.

The lemma holds as it holds in both possible scenarios. □

We denote by $\mathcal{I}^*$ the first iteration (i.e., view) such that all the correct processes output a PoD by the end of $\mathcal{I}^*$. If such an iteration does not exist, then $\mathcal{I}^* = \infty$.

Lemma 12. Let $\mathcal{I} > \mathcal{I}^*$ be an iteration of the for loop of ADA-DISP (Algorithm 5) such that every correct process has already acquired a digest-signature pair. Then, no correct process execute the sub-protocol pod_creation_attempt (line 16 of Algorithm 5).

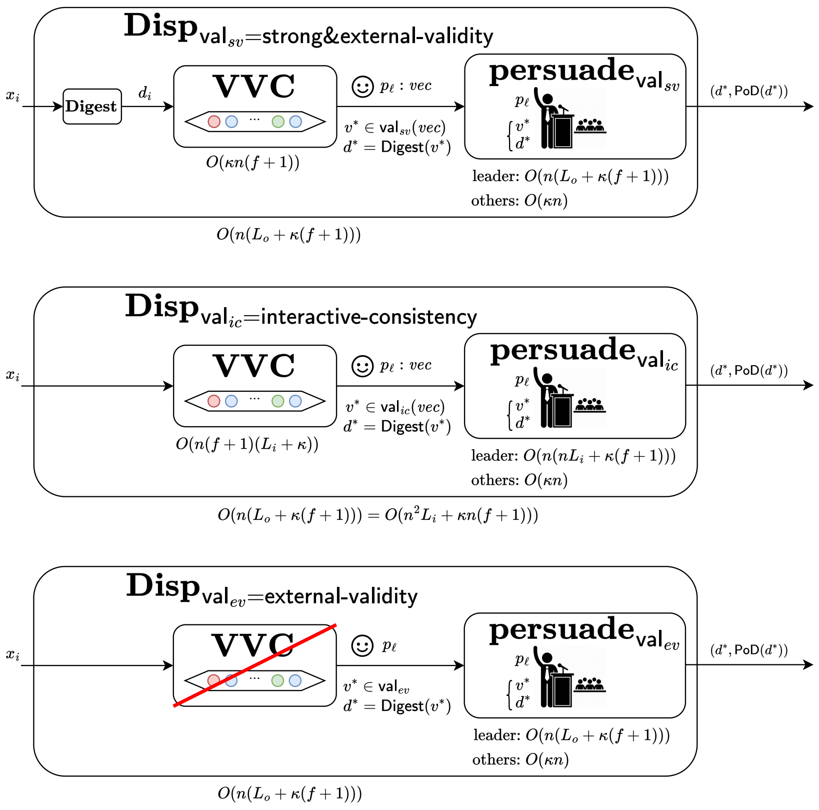

$\text{Disp}_{\mathsf{val}_{sv}=\text{strong\&external-validity}}$
x_i
Digest
d_i
VVC
$O(\kappa n(f+1))$
$p_\ell : vec$
$v^* \in \mathsf{val}_{sv}(vec)$
$d^* = \mathsf{Digest}(v^*)$
$\text{persuade}_{\mathsf{val}_{sv}}$
p_ℓ
v^*
d^*
leader: $O(n(L_o + \kappa(f+1)))$
others: $O(\kappa n)$
$(d^*, \mathsf{PoD}(d^*))$
$O(n(L_o + \kappa(f+1)))$
$\text{Disp}_{\mathsf{val}_{ic}=\text{interactive-consistency}}$
x_i
VVC
$O(n(f+1)(L_i+\kappa))$
$p_\ell : vec$
$v^* \in \mathsf{val}_{ic}(vec)$
$d^* = \mathsf{Digest}(v^*)$
$\text{persuade}_{\mathsf{val}_{ic}}$
p_ℓ
v^*
d^*
leader: $O(n(nL_i + \kappa(f+1)))$
others: $O(\kappa n)$
$(d^*, \mathsf{PoD}(d^*))$
$O(n(L_o + \kappa(f+1))) = O(n^2 L_i + \kappa n(f+1)))$
$\text{Disp}_{\mathsf{val}_{ev}=\text{external-validity}}$
x_i
VVC
p_ℓ
$v^* \in \mathsf{val}_{ev}$
$d^* = \mathsf{Digest}(v^*)$
$\text{persuade}_{\mathsf{val}_{ev}}$
p_ℓ
v^*
d^*
leader: $O(n(L_o + \kappa(f+1)))$
others: $O(\kappa n)$
$(d^*, \mathsf{PoD}(d^*))$
$O(n(L_o + \kappa(f+1)))$

Fig. 5. Different instances of ADA-DISP for some natural specific validity properties.

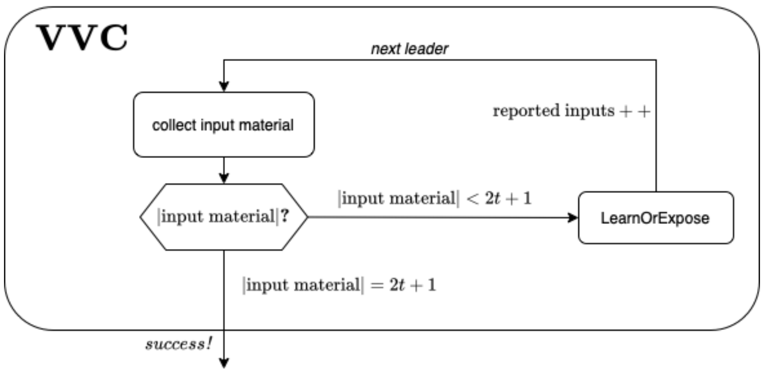


Fig. 6. **VVC** loop. When a correct leader drives the view, it either successfully obtains a vector of $2t + 1$ input materials, or triggers a LearnOrExpose instance for a newly accused process p_a. After at most $2f + 1$ views, a successful collection must happen.

PROOF. Let p_i be a correct process that acquired a digest-signature pair. Trivially, $pod_i \neq \bot$ is preserved by the end of the catch_up protocol. Hence, the check at line 15 does not pass, which implies that the protocol pod_creation_attempt is not invoked at line 16 of Algorithm 5. □

Next, we prove the communication cost of any iteration $\mathcal{I} > \mathcal{I}^*$.

Lemma 13. Let $\mathcal{I} > \mathcal{I}^*$ be an iteration of the for loop of ADA-DISP (Algorithm 5) such that every correct process has already acquired a digest-signature pair. Then, the communication complexity of this iteration is either (1) $O(\kappa f)$ if the leader is correct, or (2) $O(\kappa n)$ if the leader is Byzantine.

PROOF. By Lemma 12, the only communication to consider is the one of the catch-up phase. The case where the leader is Byzantine is straightforward as the protocol follows a leader-based paradigm, where each message has a size of at most $O(\kappa)$ bits. Assume the leader is correct; let p_ℓ be the leader. No correct process issues an AID_REQ message. Moreover, p_ℓ responds to each incoming AID_REQ message. Given that there are at most $O(f)$ such messages (originating from faulty processes), and considering that any PoD comprises one word of $O(\kappa)$ bits, p_ℓ transmits a total of $O(\kappa f)$ bits. □

Finally, we analyze the collective cost of all iterations greater than $\mathcal{I}^*$.

Lemma 14. The communication complexity of all the iterations in $(\mathcal{I}^* : n]$ of the for loop of ADA-DISP (Algorithm 5) is at most $O(\kappa n f)$.

PROOF. Among n iterations, there are (1) $n - f$ iterations with correct leaders, and (2) f iterations with faulty leaders. Hence, due to Lemma 13, the communication complexity can be expressed as

$$(n - f) \cdot O(\kappa f) + f \cdot O(\kappa n) = O(n\kappa f) + O(n\kappa f) = O(n\kappa f).$$

Therefore, the lemma holds. □

Lemma 14 allows us to focus on the cost of reaching the iteration $\mathcal{I}^*$. This is exactly the problem of verifiable vector collection. Before analyzing the protocol ADA-DISP$_{\text{val}}$, we define two helper functions: extract_vector_proof and extract_vector.

Definition 4 ((Helper) vector extraction)**.** We note extract_vector_proof and extract_vector the two helper methods that both take *discloses*, *culprits* of $\mathsf{Map}(\mathsf{Process} \longrightarrow \{0,1\}^*)$ type as input, where $|\mathit{discloses}.\mathsf{Keys}() \cup \mathit{culprits}.\mathsf{Keys}()| = n$. extract_vector_proof returns $(\sigma_1, ..., \sigma_n)$, while extract_vector returns $(e_1, ..., e_n)$ such that, for each process p_j, $(\text{DISCLOSE}, e_j, \sigma_j) = \mathit{discloses}[p_j]$ if $p_j \in \mathit{discloses}.\mathsf{Keys}()$, and $(e_j, \sigma_j) = (\bot, \mathit{culprits}[p_j])$ otherwise.

C.3 LearnOrExpose

In this subsection, we present and analyze in detail the LearnOrExpose primitive (Algorithm 6) that plays a crucial role in the dispersion. LearnOrExpose is designed to manage the collection of missing entries and to enforce the participation of all processes.

C.3.1 LearnOrExpose's Specification & Implementation. Each correct process p_i has the following read-only input parameters:

- the starting value, the proposal $\mathsf{input}(p_i)$ to ADA-DARE;
- the leader process p_ℓ;
- the accused process p_a,

and p_i writes to the following variables:

- the indirectly collected disclose messages $\mathit{indirect}_i$;
- the collected proofs-of-misbehaviour $\mathit{culprits}_i$;
- the processes $\mathit{contact}_i^{in}$ that have already requested the input of process p_i in some former LearnOrExpose instance;
- the processes $\mathit{contact}_i^{out}$ whose input have been already asked by p_i in some former LearnOrExpose instance.

The LearnOrExpose primitive guarantees the following properties:

- *Unforgeability:* For any correct process p_j with input v_j, no (correct or faulty) process obtains (i) a partial signature of DISCLOSE message associated to some value $v \neq v_j$, or (ii) a proof-of-misbehaviour against p_j ($\text{PoM}(p_j)$, with $\text{CombinedVerify}^{t+1}(\langle \text{ACCUSE}, p_j \rangle, \text{PoM}(p_j)) = $ *true*).
- *Liveness:* If p_ℓ is correct and all correct processes accuse the same process p_a, then, for every correct process p_i, $p_a \in \mathit{indirects}_i.\text{Keys}() \cup \mathit{culprits}_i.\text{Keys}()$ such that:
  - if $p_a \in \mathit{indirects}_i.\text{Keys}()$, then $\mathit{indirects}_i[p_a] = \langle \text{DISCLOSE}, e_a, \sigma_a \rangle$ with $\text{ShareVerify}_a^{t+1}(\langle \text{DISCLOSE}, e_a \rangle, \sigma_a) = \mathit{true}$;
  - if $p_a \in \mathit{culprits}_i.\text{Keys}()$, then $\mathit{culprits}_i[p_a] = \text{PoM}(p_a)$ with $\text{CombinedVerify}^{t+1}(\langle \text{ACCUSE}, p_a \rangle, \text{PoM}(p_a)) = \mathit{true}$.
- *Moderation:*
  - *Moderation-out*: If $p_a \in \mathit{contact}_i^{out}$ before the execution, then p_i does not send message to p_a. Moreover, $p_a \in \mathit{contact}_i^{out}$ by the end of the execution.
  - *Moderation-in*: If $p_j \in \mathit{contact}_i^{in}$ before the execution, then p_i does not send message to p_j. Moreover, if p_i sends a message to $p_j \notin \{p_\ell, p_a\}$ in an instance, then $p_j \in \mathit{contact}_i^{in}$ by the end of this instance.

  Unforgeability means that it is impossible to prove that a correct process (1) is misbehaving, or (2) started with an entry different from its own. Liveness means that under an honest leader, and with a commonly accused process p_a, by the end of the execution, every correct process either learns an entry signed by p_a or exposes p_a by delivering a corresponding undeniable proof-of-misbehaviour (PoM). Moderation allows an adaptive $O(nf)$ message complexity for the sequential composition of $O(f)$ instances of the LearnOrExpose primitive.

Implementation's description. As specified by the interface, processes start with a specific accused process p_a. In practice, the id of this process is communicated by the leader (see Algorithm 11), where the leader p_ℓ broadcasts (line 24) the id of the accused process p_a in a FAIL message, whose reception (line 28) triggers the invocation of line 29 of an instance of LearnOrExpose parameterized by p_ℓ and p_a. Note that, under a faulty leader, correct processes can start with a different accused process.

Essentially, the LearnOrExpose protocol (Algorithm 6) operates as follows:

- (Round 1) Request for Input: In the first round, all processes, except those that have previously contacted p_a (checked at line 10), send a REQUEST_INPUT message to p_a (line 11) to request p_a's input and update their $\mathit{contacts}_i^{out}$ list accordingly (line 12).
- (Round 2) Disclose Message: If a process receives a REQUEST_INPUT message from another process for the first time (checked at line 14), it sends back a DISCLOSE message containing its input (line 17) and updates its $\mathit{contacts}_i^{in}$ list (line 15).
- (Round 3) Accusation or Forwarding: Processes check if they have received a DISCLOSE message from p_a. If they have, the message is forwarded to the leader (line 22) and used to update $\mathit{indirects}_i$ accordingly. If not, and p_a's message is not in $\mathit{indirects}_i$, they send an accusation message to the leader (line 24).
- (Round 4) Leader's Broadcast: If the leader receives $t + 1$ ACCUSE messages against p_a, it combines these into a PoM (line 28) and broadcasts it (line 29). Alternatively, if a valid DISCLOSE message from p_a is received, the leader broadcasts this message for learning (line 31).
- (Round 5) Update Culprits or Indirects: Processes update their $\mathit{culprits}_i$ or $\mathit{indirects}_i$ maps based on the leader's broadcast, either by storing the PoM against p_a (line 34) or by updating their $\mathit{indirects}_i$ map with p_a's DISCLOSE message (line 36).

Algorithm 6 LearnOrExpose$_{\text{val}}$(p_ℓ, $start\_value_i$, p_a, $indirects_i$, $culprits_i$, $contact_i^{out}$, $contact_i^{in}$): Pseudocode for process p_i

```
1: Input Parameters:
2:     Process $p_\ell$                                              ▹ the leader
3:     Process $p_a$                                                 ▹ the accused process
4:     Value $start\_value_i$                                        ▹ The starting value
5:     Map(Process ⟶ Disclose_Msg) $indirects_i$
6:     Map(Process ⟶ T_Signature) $culprits_i$                       ▹ Processes proved guilty
7:     Set(Process) $contact_i^{out}$
8:     Set(Process) $contact_i^{in}$
9: Round 1:                        ▹ In practice, $p_a$ has been broadcasted by the leader just before
10:    if $p_a \notin contact_i^{out}$:
11:        send ⟨REQUEST_INPUT⟩ to $p_a$
12:        $contact_i^{out} \leftarrow contact_i^{out} \cup \{p_a\}$
13: Round 2:
14:    if ⟨REQUEST_INPUT⟩ is received from $p_j \notin contact_i^{in}$:
15:        $contact_i^{in} \leftarrow contact_i^{in} \cup \{p_j\}$
16:        let $entry_i$ = entry$_{\text{val}}$($start\_value_i$)     ▹ either Digest($start\_value_i$) or $start\_value_i$ depending on val
17:        send ⟨DISCLOSE, $entry_i$, ShareSign$_i^{t+1}$(DISCLOSE, $entry_i$)⟩ to $p_j$
18: Round 3:
19:    if $m_a$ = ⟨DISCLOSE, $entry_a$, ShareSign$_a^{t+1}$(DISCLOSE, $entry_a$)⟩ is received from $p_a$:
20:        $indirect_i[p_a] \leftarrow m_a$
21:    if $p_a \in indirect_i$.Keys():
22:        send $indirect_i[p_a]$ to $p_\ell$
23:    else:
24:        send ⟨ACCUSE, $p_a$, ShareSign$_i^{t+1}$(ACCUSE, $p_a$)⟩ to $p_\ell$
25: Round 4 (for leader $p_\ell$):
26:    if $p_i = p_\ell$:
27:        if ⟨ACCUSE, Process $p_a$, P_Signature $psig$⟩ is received from $t+1$ processes:
28:            let PoM($p_a$) ⟵ Combine$^{t+1}$({$sig$ | $sig$ is received in the $t+1$ received  messages})
29:            broadcast ⟨EXPOSE, PoM($p_a$)⟩
30:        else if a message $m'_a$ = ⟨DISCLOSE, $entry_a$, ShareSign$_a^{t+1}$(DISCLOSE, $entry_a$)⟩ is received:
31:            broadcast ⟨LEARN, $m'_a$⟩
32: Round 5:
33:        if ⟨EXPOSE, T_Signature PoM($p_a$)⟩ is received:
34:            $culprits_i[p_a] \leftarrow$ PoM($p_a$)
35:        else if ⟨LEARN, Disclose_Msg $m_a$⟩ is received:
36:            $indirects_i[p_a] \leftarrow m_a$
```

C.3.2 LearnOrExpose's Correctness. In this subsection, we prove the correctness of LearnOrExpose (Algorithm 6). First, we prove val-local-accuracy of LearnOrExpose (Algorithm 6), which will be crucial to prove Lemma 8.

Lemma 15 (LoE's local accuracy)**.** LearnOrExpose$_{\text{val}}$ (Algorithm 6) satisfies val-local-accuracy.

Proof. Item 2 follows directly as a correct process never signs a DISCLOSE message with an entry e that is different from its actual entry, as established at line 17 of Algorithm 6. Consider item 1 for a correct process p_a. Assume, for the sake of contradiction, that at least one correct process p_j partially signs an ACCUSE message against p_a. Necessarily, it does so at line 24 of some instance of Algorithm 6. This implies that p_a has been accused and is missing from $indirects_j$.Keys() by round 3, as per the check at line 21. This scenario occurs when process p_j has sent a REQUEST_INPUT message to p_a (in this or a previous instance) but did not receive a DISCLOSE response from p_a. However, as p_a is correct, it would have sent such a DISCLOSE message at line 17 of Algorithm 6 upon receiving the corresponding REQUEST_INPUT message. This event is impossible if both p_j and p_a are correct, leading to a contradiction. Hence, item 1 is satisfied as well. □

Second, we prove liveness of LearnOrExpose (Algorithm 6).

Lemma 16 (LoE's liveness). When the leader p_ℓ of LearnOrExpose$_{\text{val}}$ (Algorithm 6) is correct and all correct processes accuse the same process p_a, then, for each correct process p_i, p_a will be included in either $\mathit{indirects}_i$.Keys() or $\mathit{culprits}_i$.Keys().

Proof. By the end of round 1, every correct process has issued a REQUEST_INPUT message to p_a (line 11). By the completion of round 3, it is determined whether p_a belongs to $\mathit{indirects}_j$.Keys() for any correct process p_j (as per the check at line 21).

If p_a is absent from $\mathit{indirects}_j$.Keys() for every correct process p_j, each correct process then sends a partially-signed ACCUSE message against p_a to the leader in round 3 (line 24). In round 4, the leader combines these partial signatures into a $(t+1)$-threshold signature (line 28) and then broadcasts the resultant proof-of-misbehaviour to every correct process p_j. Subsequently, p_j maps p_a with this proof-of-misbehaviour in $\mathit{culprits}_j$ (line 34).

Conversely, if p_a is found in $\mathit{indirects}_j$.Keys() for some correct process p_j, then p_j relays the corresponding DISCLOSE message to the leader in round 3 (line 22). The leader, in turn, broadcasts this information in a LEARN message in round 4 (line 31), enabling each correct process p_j to associate p_a with its DISCLOSE message in $\mathit{indirects}_j$ (line 36). □

Finally, we prove two auxiliary lemmas to show the adaptive communication complexity of the ADA-DISP protocol.

Lemma 17 (Moderation-out). If $p_a \in \mathit{contact}_i^{out}$ prior to executing LearnOrExpose$_{\text{val}}$ (Algorithm 6), then p_i does not send a REQUEST_INPUT message to p_a. Additionally, p_a remains in $\mathit{contact}_i^{out}$ after the execution of the algorithm.

Proof. A correct process potentially sends a REQUEST_INPUT message to p_a only during round 1. If p_a is already in $\mathit{contact}_i^{out}$ before the algorithm starts, p_i refrains from sending a REQUEST_INPUT message to p_a in round 1, as per the condition at line 10. Furthermore, the execution of line 12 (contingent on passing the check at line 10) guarantees that p_a is in $\mathit{contact}_i^{out}$ at the end of round 1, and thus, remains so after the algorithm's execution. □

The aforementioned mechanism ensures that a correct process p_i never sends a REQUEST_INPUT message to the same process more than once.

Lemma 18 (Moderation-in). If $p_j \neq p_\ell$ is in $\mathit{contact}_i^{in}$ before executing LearnOrExpose (Algorithm 6), then p_i will not send a DISCLOSE message to p_j. Furthermore, if p_i sends a message to $p_j \notin \{p_\ell, p_a\}$ during an instance of the algorithm, then p_j will be in $\mathit{contact}_i^{in}$ by the end of that instance.

Proof. A correct process sends a DISCLOSE message to a process $p_j \neq p_\ell$ only in round 2. If p_j is already a part of $\mathit{contact}_i^{in}$ before the algorithm commences, p_i does not send a DISCLOSE message to p_j in round 2, as dictated by the check at line 14.

If p_i communicates with $p_j \notin \{p_\ell, p_a\}$ during the algorithm, this is necessarily done via a DISCLOSE message sent in round 2. This action triggers line 15, leading to p_j being included in $\mathit{contact}_i^{in}$ at the end of round 2 and, consequently, at the end of that particular instance. □

The mechanism above ensures that a correct process p_i never sends a DISCLOSE message to the same process more than once.

C.4 Persuasion: Problem Definition

In this subsection, we briefly discuss the Persuasion problem. We present solutions for instances of this problem parameterized with val_{ic} and val_{sv} in Appendix C.5 and Appendix C.6, respectively.

Persuasion is a problem that effectively represents the final step to solve the dispersion problem (after having achieved verifiable vector collection). Hence, it naturally shares strong similarities with the dispersion problem.

Persuasion is a problem parameterized by (1) a validity property val, and (2) a vector predicate vec_pred for val. In practice, we will only consider the vector predicate $\mathsf{VerifyVector}_{\mathsf{val}}$. Moreover, the persuasion problem is parameterized by a fixed leader p_ℓ. In an ideal scenario, the leader p_ℓ starts with (1) a vector that matches the current (i.e., active) input configuration c, i.e., entries of all the correct processes, and (2) a vector proof for the vector. If the leader is correct and the required preconditions are met, the postconditions of the dispersion problem are satisfied. Otherwise, the safety properties of the dispersion problem are secured, i.e., the safety of dispersion is always preserved.

The persuasion problem requires the following properties to hold except with negligible probability $neg(\kappa)$:

- *Integrity:* If a correct process triggers $\mathsf{acquire}(d, \sigma^d)$, then $\mathsf{CombinedVerify}^{t+1}(\langle \textsc{ack}, d\rangle, \sigma^d))$ $= true$.
- *Redundancy:* Let a correct process obtain (d, σ^d) such that $\mathsf{CombinedVerify}^{t+1}(\langle \textsc{ack}, d\rangle, \sigma^d) =$ $true$, for some digest d and some threshold signature σ^d. Then, there exists a pre-image v of d, verifying $\mathsf{Digest}(v) = d$, such that:
  - Observation: (at least) $t + 1 - f$ correct processes have obtained v;
  - Validity: $v \in \mathsf{val}(c)$.

  Let us remark that Validity might imply $\mathsf{externallyValid}(v) = true$ depending on val.
- *Optimistic termination:* If the leader is correct and starts with a pair (vec, vec_proof) such that $\mathsf{vec\_pred}(vec, vec\_proof) = true$, then every correct process acquires at least one digest-signature pair.

C.5 Persuasion for Interactive Consistency

In this subsection, we present $\mathsf{persuade}_{\mathsf{val}_{ic}}$ (Algorithm 7) that solves the persuasion problem parameterized with val_{ic} and $\mathsf{VerifyVector}_{\mathsf{val}_{ic}}$.

Let us explain our approach. Assume the leader p_ℓ is correct and stores $(discloses_\ell, culprits_\ell)$ such that $\mathsf{VerifyVector}_{\mathsf{val}_{ic}}(vec, proof) = true$ with $vec = (e_1, ..., e_n) = \mathsf{extract\_vector}(discloses_\ell, culprits_\ell)$ and $proof = \mathsf{extract\_vector\_proof}(discloses_\ell, culprits_\ell)$. In such a situation, p_ℓ extracts the pair $(vec, proof)$ (line 8), and broadcasts it (line 9). Upon reception of the pair (line 11), every correct process acknowledges the pair (line 12), thus allowing the leader to (1) receive $t + 1$ partially signed ACK messages (line 14), and (2) generate a PoD (line 15) that is then disseminated to every process (line 16), delivered (line 18), and returned by every correct process (line 19). In the event that the leader is faulty, the process may not receive any PoD and returns $\perp$ instead (line 21).

We now prove the local accuracy property of $\mathsf{persuade}_{\mathsf{val}_{ic}}$, which will be used later to prove the local accuracy property of $\textsc{ada-Disp}_{\mathsf{val}_{ic}}$ and, thus, the global accuracy of $\textsc{ada-Disp}_{\mathsf{val}_{ic}}$ and unforgeability of $\mathsf{VerifyVector}_{\mathsf{val}_{ic}}$.

Lemma 19. $\mathsf{persuade}_{\mathsf{val}_{ic}}$ (Algorithm 7) satisfies val_{ic}-local-accuracy.

Proof. Items 1 and 2 are trivially verified since no correct process signs an ACCUSE message or a DISCLOSE message in $\mathsf{persuade}_{\mathsf{val}_{ic}}$. □

Algorithm 7 $\text{persuade}_{\text{val}_{ic}}(start\_value_i, p_\ell, discloses_i, culprits_i)$: Pseudocode (for process p_i)

```
1: Input Parameters:
2:     Value $start\_value_i$                                  ▷ the input value; first parameter
3:     Process $p_\ell$                                        ▷ the leader; second parameter
4:     Map(Process → Disclose_Msg) $discloses_i$               ▷ storing values directly or indirectly obtained
5:     Map(Process → T_Signature) $culprits_i$                 ▷ storing PoM indirectly obtained
6: Round 1:                                                    ▷ executed only by $p_\ell$
7:     if $p_i = p_\ell$:
8:         let $vec \leftarrow$ extract_vector($discloses_\ell$, $culprits_\ell$), $proof \leftarrow$ extract_vector_proof($discloses_\ell$, $culprits_\ell$)).
9:         broadcast ⟨VECTOR, $vec$, $proof$⟩
10: Round 2:
11:     if a ⟨VECTOR, $vec$, $proof$⟩ message is received from $p_\ell$ such that $\text{VerifyVector}_{\text{val}_{ic}}(vec, proof)$:
12:         send ⟨VEC_ACK, $\text{ShareSign}_i^{t+1}$(ACK, Digest($vec$))⟩ to $p_\ell$    ▷ allow the leader to create a succinct PoD
13: Round 3:
14:     if $p_i = p_\ell$ and ⟨VEC_ACK, P_Signature $sig$⟩ is received from $t+1$ processes:    ▷ the leader creates a succinct PoD
15:             let $pod \leftarrow \text{Combine}^{t+1}$({$sig$ | $sig$ is received in the $t+1$ received VEC_ACK messages})
16:             broadcast ⟨COMPACT_CERT, Digest($vec$), $pod$⟩
17: Round 4:
18:     if ⟨COMPACT_CERT, Digest $d$, PoD $\sigma$⟩ is received from $p_\ell$ such that $\text{CombinedVerify}^{t+1}$(⟨ACK, $d$⟩, $\sigma$):
19:         return $d, \sigma$
20:     else:
21:         return $\bot, \bot$
```

Lemma 20. *If protocol* $\text{persuade}_{\text{val}_{ic}}$ *(Algorithm 7) is a sub-routine of* ADA-DISP$_{\text{val}_{ic}}$*, then it solves the Persuasion problem parameterized with* val_{ic} *and* $\text{VerifyVector}_{\text{val}_{ic}}$*.*

PROOF. Assume some correct process outputs a pair $(d_i, \text{PoD}(d_i))$. Necessarily the check at line 18 has passed, which implies $\text{CombinedVerify}^{t+1}(\langle \text{ACK}, d_i\rangle, \text{PoD}(d_i)) = true$, and thus the integrity property is satisfied. Moreover, it implies that a set S of at least $t+1-f$ correct processes partially signed the message $\langle \text{ACK}, d_i\rangle$. It means all the members of S partially-signed the message $\langle \text{ACK}, d_i\rangle$ at line 12, upon the reception of a vector vec and a corresponding valid vector proof at line 11. This implies the observation property. This also implies the validity property, since such a proof is unforgeable by Lemma 10.

Let us prove optimistic termination. Suppose that the leader p_ℓ is correct and p_ℓ initially stores the pair $(discloses_\ell, culprits_\ell)$ such that $\text{VerifyVector}_{\text{val}_{ic}}(vec, proof) = true$ with $vec = \text{extract\_vector}(discloses_\ell, culprits_\ell) = (e_1, ..., e_n)$ and $proof = \text{extract\_vector\_proof}(discloses_\ell, culprits_\ell)$. The pair $(vec, proof)$ is computed (line 8), and then broadcast (line 9). Upon its reception (line 11), every correct process acknowledges it (line 12), allowing the reception of $t+1$ partially signed ACK messages by the leader (line 14). The leader then generate a PoD (line 15), which is then broadcast (line 16) and delivered by every correct process (line 18) and returned by each of them (line 19). □

C.6 Persuasion for Strong Validity

This subsection presents $\text{persuade}_{\text{val}_{sv}}$ (Algorithm 8), an efficient solution for the persuasion problem parameterized with val_{ic} and $\text{VerifyVector}_{\text{val}_{ic}}$, where val_{sv} = strong&external_validity. This protocol relies on two sub-protocols: partition (Algorithm 9) and construct_strong_negative_certificate (Algorithm 10). Recall that in this validity, an entry is in the form of digest (i.e. $\text{Entry}_{\text{val}_{sv}}$ = Digest).

Algorithm 8 handles two types of certificates (inspired from Big Buckets [67]):

- *Positive certificates:* Positive certificates vouch that one specific value is safe. Namely, a certificate Σ^+ is said to be a *positive bucket certificate* for a digest d if and only if Σ^+ is a $(t+1)$-threshold signature for d where d is a digest of some processes' input value. Formally,

$\mathsf{CombinedVerify}^{t+1}(\langle \textsc{disclose}, d \rangle, \Sigma^{+}) = \mathit{true}$. Since each correct process starts with an externally valid value v and a positive bucket certificate is a $(t+1)$-threshold signature for some input value's digest, the digest d associated with a positive bucket certificate necessarily has an externally valid pre-image v that has been obtained by $t+1-f$ correct processes. Hence, a positive bucket certificate acts as a special PoD.

- *Negative certificates:* Negative certificates (see figure 7) prove that not all correct processes have started with the same value. However, a *negative bucket certificates* involves additional cryptographic materials compared to its counterpart in [67]. In order to properly define negative bucket certificates, we introduce *extended groups*. Intuitively, an extended group $g = (x, y, \_)$ is a cryptographic object that allows to convince that $T + \hat{T} \geq t+1$ processes started with a value whose digest is outside a certain segment $[x, y]$. Thus, a tuple of extended groups whose associated segments form a partition of the entire domain of possible digests constitutes a proof that the correct processes did not start with the same value. In the description, T represents a contribution obtained from bona fide processes, with a compact representation, while $\hat{T}$ represents a contribution obtained indirectly via the LearnOrExpose primitive. More formally, an extended group g is a tuple $(x, y, T, \sigma, \mathit{indirects}, \mathit{culprits})$, such that:
  1. $x, y \in \mathsf{Digest\_Value}$ with $x \leq y$,
  2. *indirects* is a dictionary mapping process ids to a $\textsc{dicslose}$ message, and for each $p_j \in \mathit{indirects}.\mathsf{Keys}()$, $\mathit{indirects}[p_j] = \langle \textsc{disclose}, e_j, \mathsf{ShareSign}_j^{t+1}(\textsc{disclose}, e_j) \rangle$ with $e_j \notin [x, y]$,
  3. *culprits* is a dictionary mapping process ids to a proof-of-misbehaviour, and for each $p_j \in \mathit{culprits}.\mathsf{Keys}()$, $\mathit{culprits}[p_j] = \mathsf{PoM}(p_j)$.
  4. $U \triangleq (\Pi \setminus \mathit{indirects}.\mathsf{Keys}()) \setminus \mathit{culprits}.\mathsf{Keys}()$ (U is a universe),
  5. $T \in [1, t+1]$ is a threshold,
  6. σ is a (T, U)-MTS of (x, y), i.e.
     $\mathsf{Verify}((x, y), \sigma, T, \mathsf{VK}_U) = \mathit{true}$ with $(CK_U, VK_U) = \mathsf{UniverseSetup}(U)$,
  7. $(\mathit{indirects}.\mathsf{Keys}() \cup \mathit{culprits}.\mathsf{Keys}()) \cap U = \emptyset$,
  8. $|\mathit{indirects}.\mathsf{Keys}() \cup \mathit{culprits}.\mathsf{Keys}()| = \widehat{T}$ such that $T + \widehat{T} \geq t+1$,

  Lastly, a negative bucket certificate is a tuple $\left(g_1 = (x_1, y_1, \_), \dots, g_k = (x_k, y_k, \_)\right)$ of $k \leq 3$ extended groups, such that:
  - $x_{k'+1} = y_{k'} + 1$ for each $k' \in [1 : k-1]$[18],
  - $x_1 = d_{min}$ (the first digest in the lexicographic order), and
  - $y_k = d_{max}$ (the last digest in the lexicographic order)

  Note that for each $d \in \mathsf{Digest\_Value}$, there is exactly one i such that $x_i \leq d \leq y_i$.

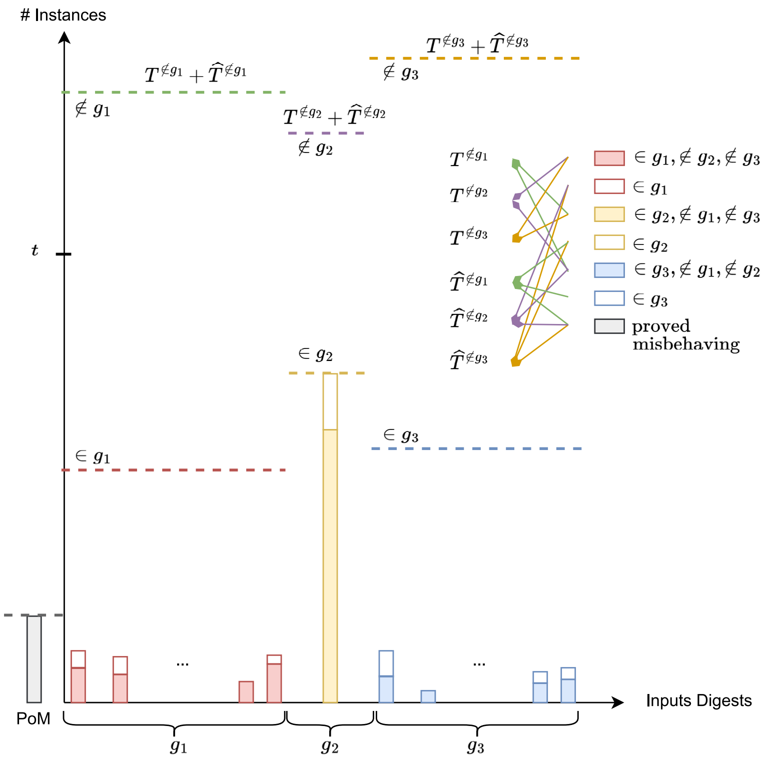
We now explain Algorithm 8 in more details.

When a correct leader p_ℓ engages in $\mathsf{persuade}_{\mathsf{val}_{sv}}$ with val_{sv} = strong&external_validity, it verifies whether it has received a common digest from at least $t+1$ processes (line 12). If affirmative, p_ℓ builds a positive bucket certificate by combining the received partial signatures into a $(t+1)$-threshold signature (line 14). Otherwise, it partitions the received digests into groups via Algorithm 9 (line 18) such that:

- Each constructed group $g = (x, y, \cdot, \cdot, \cdot, \cdot)$ includes the digests of at most t processes.
- For any two contiguous groups, the digests of at least $t+1$ processes are included in their union.

Crucially, p_ℓ manages to partition all received digests into at most $3 \in O(1)$ groups. Upon finalizing the groups, p_ℓ communicates them to all processes (line 19). Receiving these groups, p_i responds

[18]+1 returns the next value in lexicographic order



# Instances
$T^{\notin g_1} + \widehat{T}^{\notin g_1}$
$\notin g_1$
$T^{\notin g_2} + \widehat{T}^{\notin g_2}$
$\notin g_2$
$T^{\notin g_3} + \widehat{T}^{\notin g_3}$
$\notin g_3$
t
$T^{\notin g_1}$
$T^{\notin g_2}$
$T^{\notin g_3}$
$\widehat{T}^{\notin g_1}$
$\widehat{T}^{\notin g_2}$
$\widehat{T}^{\notin g_3}$
$\in g_1, \notin g_2, \notin g_3$
$\in g_1$
$\in g_2, \notin g_1, \notin g_3$
$\in g_2$
$\in g_3, \notin g_1, \notin g_2$
$\in g_3$
proved misbehaving
$\in g_2$
$\in g_3$
$\in g_1$
...
...
PoM
Inputs Digests
g_1
g_2
g_3

Fig. 7. Negative bucket certificate. The entries partitioned by the leader (line 18) into at most $3 \in O(1)$ extended groups are represented in red, yellow and blue on the figure. Processes that consistently reply (line 25), with a (multiverse) partial signature (line 24) for each extended group to which processes do not belong to (see the check at line 23) contribute to the (multiverse) threshold $T^{\notin g}$ for the different extended groups. Byzantine processes that do not consistently reply, will indirectly contribute to $\hat{T}^{\notin g}$, either by a proof-of-misbehaviour or by a corresponding DISCLOSE message, potentially indirectly obtained via a LearnOrExpose instance. For each extended group g, $T^{\notin g} + \hat{T}^{\notin g} \geq t + 1$ proves that some correct process did not start with an entry in g.

with MTS partial signatures for each group to which it does not belong (lines 24 and 25). Notice that here, p_i can sign an MTS partial signature without any knowledge of what the universe and threshold would be.

After accumulating MTS partial signatures from other processes (line 28), p_ℓ builds a negative bucket certificate via Algorithm 10 (line 29). If the leader p_ℓ has previously created a positive bucket certificate, this segment is bypassed. If p_ℓ has accounted for $2t + 1$ processes at line 16, the boolean variable *negative_cert_ready* is set to true at line 17, allowing the check at line 27 to pass. This is followed by the generation of a negative certificate at line 29, which is then broadcast at line 31.

On receiving a certificate, p_i engages in two additional rounds (rounds 4 and 5) to construct a succinct PoD, in the form of a simple $(t + 1)$-threshold signature. Finally, if p_i acquires a valid succinct PoD from p_ℓ (line 42), it returns that PoD (line 43). In the absence of such a PoD, p_i returns $\perp$ (line 45) instead.

For completeness, we will explain the partitioning logic in Algorithm 9. In a nutshell, the algorithm will first sort the digests (line 9). It then iterates through the sorted digests and greedily forms the groups (line 17). If the current digest can be added to the last created group such that the group still contain at most t digests (line 18), then the current digest will be added to that group. Otherwise, a new group will be created, containing the current digest (line 21). Finally, the algorithm returns the formed groups (line 28).

We now prove that the partitioning done in Algorithm 9 will produce at most $3 \in O(1)$ groups.

Lemma 21. Consider a correct process p_ℓ which has received at most $2t + 1$ entries and no entry is received more than t times. The execution of partition (Algorithm 9) by process p_ℓ returns a partition of $\mathsf{Entry}_{\mathsf{val}}$ of at most $3 \in O(1)$ groups such that (1) each group contains at most t received values, and (2) any two adjacent groups contain at least $t + 1$ received values.

Proof. To prove the lemma, we reuse the argument from [67]. As any two adjacent groups contain (collectively) at least $t + 1$ values, and there are $2t + 1$ values, there can be at most $3 \in O(1)$ groups. Indeed, the first two groups contain (at least) $t + 1$ values. Assuming there are at least 4 groups, the same would also hold for the third and fourth groups. This contradicts the fact that l received at most $2t + 1$ values. Hence, there are at most $3 \in O(1)$ groups, which concludes the lemma. □

We need to prove the accuracy of $\mathsf{persuade}_{\mathsf{val}_{sv}}$, which later will be used to prove the accuracy of ADA-DISP$_{\mathsf{val}_{sv}}$.

Lemma 22. $\mathsf{persuade}_{\mathsf{val}_{sv}}$ (Algorithm 8) satisfies val_{sv}-local-accuracy.

Proof. Items 1 and 2 are trivially verified since no correct process ever signs an ACCUSE message or a DISCLOSE message in $\mathsf{persuade}_{\mathsf{val}_{sv}}$. □

Lemma 23 (Unforgeability of negative certificate). If protocol $\mathsf{persuade}_{\mathsf{val}_{sv}}$ (Algorithm 8) is a subroutine of ADA-DISP$_{\mathsf{val}_{sv}}$, Then no negative bucket certificate can be forged in $\mathcal{E}' \in execs(\mathsf{persuade}_{\mathsf{val}_{sv}})$ such that $\mathcal{E}'$ corresponds to some input configuration c, where there exists $v \in \mathsf{Value}_I : \forall p_i \in \pi(c), \mathsf{proposal}(c[i]) = v$.

Proof. By contradiction, suppose all correct processes start with the same value v. Assume $\Sigma^- = (g_1, \dots, g_m)$ is a negative bucket certificate where $m \leq 3$ and each $g_k = (x_k, y_k, T_k, \sigma_k, indirects^{g_k}, culprits)$ for $1 \leq k \leq m$. It follows that for some $k \in [1 : m]$, $e \triangleq \mathsf{entry}_{\mathsf{val}_{sv}}(v)$ falls within $[x_k, y_k]$. Hence, a correct process p_j must either (i) be in $indirect^{g_k}.\mathsf{Keys}()$ and have signed (DISCLOSE, e_j) with $e_j \notin [x_k, y_k]$, or (ii) be in $culprits.\mathsf{Keys}()$, or (iii) be part of $U^{g_k} \triangleq (\Pi \setminus indirect^{g_k}.\mathsf{Keys}()) \setminus culprits.\mathsf{Keys}()$ and have signed (x_k, y_k). Scenario (i) is not possible because $e \triangleq \mathsf{entry}_{\mathsf{val}_{sv}}(v) \in [x_k, y_k]$ and item 2 of Definition 1 of val_{sv}-local-accuracy, ensured by Lemma 8. Scenario (ii) cannot occur as it would contradict the item 1 of Definition 2 of val_{sv}-global-accuracy, ensured by Lemma 9. Scenario (iii) is also not possible because p_j started with entry $e \in [x_k, y_k]$, and a correct process only signs groups that it does not belong to (see lines 23 and 24 of Algorithm 8). □

We will prove the 'correctness' of Algorithm 10.

Lemma 24. Assume a correct process executes Algorithm 10 with parameters satisfying the following conditions:

(1) $\mathsf{VerifyVector}_{\mathsf{val}_{sv}}(vec, proof)$ where $vec = (e_1, ..., e_n)$, with $(vec, proof) = (\mathsf{extract\_vector}(discloses, culprits), \mathsf{extract\_vector\_proof}(discloses, culprits))$
(2) $groups = \mathsf{partition}(discloses)$,
(3) For each $p_i \in replies.\mathsf{Keys}()$, $replies[p_i] = \langle \text{PARTITION\_REPLY}, \{(x_g, y_g, \_), \mathsf{Sign}_i(x_g, y_g)\}_{g \in G_i} \rangle$, where G_i is a proper subset of $groups$ with $|G_i| = |groups| - 1$ and for all $g \in G_i$, $e_i \notin [x_g, y_g]$.

Under these conditions, Algorithm 10 outputs a negative bucket certificate.

Proof. Given that initially, $groups = \text{partition}(discloses)$, every g in $groups$ satisfies item 1, and collectively, these members form a partition.

Consider an extended group $g = (x^g, y^g, |J^g|, \sigma^g, indirect^g, culprit)$ in $groups$. Item 2 is satisfied due to the check at line 10 in algorithm 10. Item 3 is inherently met by definition. Item 4 is a definition. The integer $|J^g|$ meets the requirement in item 5. All signers in J^g are in $U^g \triangleq (\Pi \setminus indirects^g.\text{Keys}()) \setminus culprits.\text{Keys}()$, so $\sigma^g = \text{Combine}(\text{CK}_{U^g}, S^g_{J^g})$ with $(\text{CK}_{U^g}, \text{VK}_{U^g}) \leftarrow \text{UniverseSetup}(U^g)$ fulfills item 6.

Item 7 is fulfilled through the construction of U^g, which excludes $indirects^g.\text{Keys}()$ and $culprits.\text{Keys}()$. What remains is to demonstrate that item 8 is fulfilled. As initially, $groups = \text{partition}(discloses)$, $discloses$ includes at least $t + 1 - |culprits.\text{Keys}()|$ DISCLOSES messages of the form $discloses[p_j] = \langle \text{DISCLOSE}, e_j, \text{ShareSign}^{t+1}_j(\text{Disclose}, e_j)\rangle$ with $e_j \notin [x^g, y^g]$. Coupling this with line 11, preceded by the check at line 10, item 8 is verified. □

Lemma 25. If protocol $\text{persuade}_{\text{val}_{sv}}$ (Algorithm 8) is a sub-routine of ADA-DISP$_{\text{val}_{sv}}$, then it solves the Persuasion problem parameterized with val_{sv} and $\text{VerifyVector}_{\text{val}_{sv}}$.

Proof. Assume some correct process outputs the pair $(d_i, \text{PoD}(d_i))$. The check at line 42 must have passed, which means $\text{CombinedVerify}^{t+1}(\langle \text{ACK}, d_i\rangle, \text{PoD}(d_i)) = true$. Thus, integrity is verified. Moreover, it implies that a set S of at least $t + 1 - f$ correct processes partially signed the message $\langle \text{ACK}, d_i\rangle$. It must have been done either at line 36 upon the reception of a valid positive bucket certificate, or at line 34, upon the reception of a negative bucket certificate. Assume one member of S signed the message $\langle \text{ACK}, d_i\rangle$ at line 36 upon the reception of a valid positive bucket certificate. Then, a set S' of at least $t + 1 - f$ correct processes have partially-signed the message $\langle \text{DISCLOSE}, d\rangle$, where d is necessarily their common entry. By Lemma 8, this implies observation and validity (and thus, redundancy). Assume no member of S partially-signed the message $\langle \text{ACK}, d_i\rangle$ at line 36 upon the reception of valid positive bucket certificate. It means all the members of S partially-signed the message $\langle \text{ACK}, d\rangle$ at line 34 upon the reception of a negative bucket certificate, associated with some value v such that $\text{Digest}(v) = d$. This implies observation. Strong validity is guaranteed by Lemma 23. Moreover, v is externally valid since the check line 33 passes for at least one correct process. Thus, validity is also ensured in this case and thus, redundancy is also ensured.

Let us prove optimistic termination. Assume the leader p_ℓ is correct and initially stores $discloses_\ell$ and $culprits_\ell$ such that $\text{VerifyVector}_{\text{val}_{sv}}(vec, proof)$ with $vec = (e_1, ..., e_n) = \text{extract\_vector}(discloses_\ell, culprits_\ell)$ and $proof = (\sigma_1, ..., \sigma_n) = \text{extract\_vector\_proof}(discloses_\ell, culprits_\ell)$. If the same entry e appears $t + 1$ times, a positive bucket certificate is built at line 14 and broadcast at line 15. Upon its reception at line 35, every correct process replies with an acknowledgement at line 36, allowing the reception of $t + 1$ partially signed ACK messages by the leader (line 38). The leader then combines them into a PoD using $(t + 1)$-threshold signature (line 39), which is then broadcast at line 40. Every correct process will receive the PoD at line 42 and use it as the return value (line 43). Hence, in this case, optimistic termination holds.

If no entry e appears $t + 1$ times, using Algorithm 9, p_ℓ organizes all received values into $O(1)$ groups (line 18), and then communicates these groups to all processes via PARTITION_REQ messages (line 19). Upon the reception of a PARTITION_REQ message (line 21), each correct process replies with a consistent PARTITION_REPLY message (line 25). Upon the reception of $2t + 1 - f$ PARTITION_REPLY messages (line 28), the necessary pre-conditions of Lemma 24 for generating a negative bucket certificate via Algorithm 10 are met. The generated negative bucket certificate (line 29) is then broadcast at line 31. Upon its reception (line 33), every correct process replies with an acknowledgement at line 34. This ensures the reception of $t + 1$ partially-signed ACK message by the leader at line 38. The leader then combines them into a PoD using $(t + 1)$-threshold signature

Algorithm 8 $\text{persuade}_{\text{val}_{sv}}(start\_value_i, p_\ell, dicloses_i, culprits_i)$: Pseudocode (for process p_i)

```
1:  Input Parameters:
2:      Process $p_\ell$                                          ▷ the leader; first parameter
3:      Value $start\_value_i$                                     ▷ the input value; second parameter
4:      Map(Process → Disclose_Msg) $discloses_i$                  ▷ storing values directly or indirectly obtained
5:      Map(Process → T_Signature) $culprits_i$                    ▷ storing PoM indirectly obtained
6:  Variables:
7:      Set(Extended_Group) $groups_i \leftarrow \bot$             ▷ partitioning groups
8:      Boolean $negative\_cert\_ready \leftarrow false$            ▷ attest if the leader will have to produce a negative certificate
9:      Digest_Value $certified\_digest_i \leftarrow \bot$         ▷ the certified digest
10: Round 1:
11:     if $p_i = p_\ell$:
12:         if exists Digest_Value $d$ such that $|\{j \in [1:n] | discloses_i[p_j] = \langle \text{DISCLOSE}, d, \text{P\_Signature } sig \rangle\}| \geq t+1$:
13:             $certified\_digest_i \leftarrow d$
14:             let $positive\_cert \leftarrow \text{Combine}^{t+1}(\{sig \mid sig$ is received in the $t+1$ received DISCLOSE messages$\})$
15:             broadcast $\langle \text{POS\_CERT}, d, positive\_cert \rangle$        ▷ a positive certificate has been built and broadcasted
16:         else if $|discloses_i.\text{Keys}() \cup culprits_i.\text{Keys}()| = 2t+1$:     ▷ a precondition if honestly triggered
17:             $negative\_cert\_ready \leftarrow true$
18:             $groups_i \leftarrow \text{partition}(discloses_i)$
19:             broadcast $\langle \text{PARTITION\_REQ}, groups_i \rangle$
20: Round 2:
21:     if $\langle \text{PARTITION\_REQ}, \text{Set(Extended\_Group)} \ groups \rangle$ is received from $p_\ell$ such that $|groups| \in O(1)$:
22:         let $negatives \leftarrow \emptyset$
23:         for each $(g = (x, y, \cdot, \cdot, \cdot, \cdot)) \in groups$ such that $\neg(x \leq \text{Digest}(start\_value_i) \leq y)$:
24:             $negatives \leftarrow negatives \cup (g, \text{Sign}_i(x, y))$        ▷ attest its digest is not included in $g$
25:         send $\langle \text{PARTITION\_REPLY}, negatives \rangle$ to $p_\ell$
26: Round 3:
27:     if $p_i = p_\ell$ and $negative\_cert\_ready = true$:
28:         let $replies$ be the set of received PARTITION_REPLY messages that are consistent with the sender's disclose
29:         let $neg\_cert \leftarrow \text{construct\_strong\_negative\_certificate}(replies, groups_i, discloses_i, culprits_i)$
30:         $certified\_digest_i \leftarrow \text{Digest}(start\_value_i)$
31:         broadcast $\langle \text{NEG\_CERT}, start\_value_i, neg\_cert \rangle$
32: Round 4:
33:     if a valid $\langle \text{NEG\_CERT}, \text{Value } value, \text{Certificate } \Sigma \rangle$ message is received from $p_\ell$ such that externallyValid($value$):
34:         send $\langle \text{ACK}, \text{ShareSign}_i^{t+1}(\text{ACK}, \text{Digest}(value)) \rangle$ to $p_\ell$      ▷ allows the leader to create a succinct PoD
35:     else if a valid $\langle \text{POS\_CERT}, \text{Digest } d, \text{PC } \sigma \rangle$ message is received from $p_\ell$ s.t. $\text{CombineVerify}^{t+1}(\langle \text{DISCLOSE}, d \rangle, \sigma)$:
36:         send $\langle \text{ACK}, \text{ShareSign}_i^{t+1}(\text{ACK}, d) \rangle$ to $p_\ell$      ▷ positive certificate proves correct observation of $\text{Digest}^{-1}(d)$
37: Round 5:
38:     if $p_i = p_\ell$ and $\langle \text{ACK}, \text{P\_Signature } sig \rangle$ is received from $t+1$ processes:      ▷ The leader creates a succinct PoD
39:         let $pod \leftarrow \text{Combine}^{t+1}(\{sig \mid sig$ is received in the $t+1$ received ACK messages$\})$
40:         broadcast $\langle \text{COMPACT\_CERT}, certified\_digest_i, pod \rangle$
41: Round 6:
42:     if $\langle \text{COMPACT\_CERT}, \text{Digest\_Value } d, \text{PoD } \sigma \rangle$ is received from $p_\ell$ such that $\text{CombineVerify}^{t+1}(\langle \text{ACK}, d \rangle, \sigma)$:
43:         return $d, \sigma$
44:     else:
45:         return $\bot, \bot$
```

(line 39), which is then broadcast at line 40 . Every correct process will receive the PoD at line 42 and use it as the return value at line 43. Thus, optimistic termination also holds in this case and therefore, optimistic termination always holds. □

Algorithm 9 partition(*discloses*): Revised Partitioning Algorithm

```
1: Input Parameter:
2:     Map(Process → Digest_Value) discloses                ▷ map of process to its disclosed entry
3: Variables:
4:     List(Extended_Group) groups ← []                    ▷ list of groups for partitioning
5:     SortedList(Digest_Value) digests ← []               ▷ sorted list of digests
6:     Map(Digest_Value → Integer) frequency ← empty        ▷ frequency of each digest
7: Preparation:
8: for each ⟨DISCLOSE, $e_j$, ShareSign$_j^{t+1}$(DISCLOSE, $e_j$)⟩ in discloses.Values():
9:     digests.insert($e_j$)                                ▷ Keep the list sorted
10:    if $e_j \notin$ frequency.Keys():
11:        frequency[$e_j$] = 1
12:    else:
13:        frequency[$e_j$] = frequency[$e_j$] + 1   ▷ Map each digest to its frequency among the disclose messages
14: Partitioning Logic:
15: initialize current group as $g \leftarrow (d_{min}, d_{min}, \_)$
16: initialize value count for the current group as count ← 0
17: for each $d$ in digests:
18:    if count + frequency[$d$] $\leq t$:
19:        update the upper bound $g[2]$ to $d$
20:        increment count by frequency[$d$]
21:    else:
22:        let $d' \leftarrow g[2] + 1$
23:        add $g$ to groups
24:        reset $g \leftarrow (d', d, \_)$
25:        reset count ← frequency[$d$]
26: update the upper bound $g[2]$ with $d_{max}$
27: add $g$ to groups
28: return groups
```

C.7 Verifiable Vector Collection

In this subsection, we present how the composition of pod_creation_attempt (Algorithm 11) and ADA-DISP (Algorithm 5) solves verifiable vector collection.

C.7.1 Pseudocode of pod_creation_attempt. First, we present pod_creation_attempt (Algorithm 11) that corresponds to the attempt phase of a view of ADA-DISP. The input parameters for the protocol pod_creation_attempt include: (1) *start_value$_i$*, the value with which a process p_i initiates ADA-DISP.propose(·); (2) p_ℓ, the leader of the particular pod_creation_attempt invocation; (3) *indirects$_i$*, the disclose messages indirectly gathered through LearnOrExpose instances; (4) *culprits$_i$*, the processes identified as guilty with associated proofs-of-misbehaviour from LearnOrExpose; (5) *contact$_i^{in}$*, the processes that have previously requested p_i's entry in LearnOrExpose; and (6) *contact$_i^{out}$*, processes whose input p_i has inquired about in earlier LearnOrExpose instances.

When a correct process p_i engages in pod_creation_attempt, it initially informs the leader p_ℓ of its entry *entry$_i$* (either *start_value$_i$* or Digest(*start_value$_i$*), depending on the validity property) by sending a DISCLOSE message to p_ℓ (line 15). This message contains its partially signed entry *entry$_i$*. Upon receiving the DISCLOSE messages, including those indirectly acquired through past LearnOrExpose instances, p_ℓ (if correct) verifies whether it can link $2t + 1$ processes to a DISCLOSE message or a proof-of-misbehaviour (line 20), by also considering messages indirectly obtained via LearnOrExpose. If this cannot be achieved, p_ℓ must be able to identify a process p_a that it can accuse. p_ℓ then indicates p_a for accusation (line 24), suggesting processes to initiate LearnOrExpose (line 29) to ascertain p_a's input or expose p_a.

Algorithm 10 construct_strong_negative_certificate(*replies*, *groups*, *discloses*, *culprits*): Strong Negative Certificate Construction

```
1: Input Parameters:
2:     List(Partition_Reply_Msg) replies                    ▷ replies containing partial signatures
3:     List(Extended_Group) groups                          ▷ list of extended groups from partitioning
4:     Map(Process → Disclose_Msg) discloses                ▷ direct and indirect disclose messages
5:     Map(Process → T_Signature) culprits                  ▷ A PoM for each culprit
6: Let $I$ be the set of processes in disclose.Keys() who did not reply consistently
7: for each extended group $g = (x, y, \cdot, \cdot, \cdot)$ in groups do
8:     Map(Process → Disclose_Msg) $indirects^g \leftarrow empty$
9:     for each $p_j \in I$ do
10:        if $discloses[p_j] = \langle \text{DISCLOSE}, e_j, \text{ShareSign}_j^{t+1}(\text{Disclose}, e_j) \rangle$ with $e_j \notin [x, y]$ then
11:            $indirects^g[p_j] \leftarrow discloses[p_j]$
12:        end if
13:    end for
14:    $U^g \leftarrow (\Pi \setminus indirects^g.\text{Keys}()) \setminus culprits.\text{Keys}()$
15:    $(\text{CK}_{U^g}, \text{VK}_{U^g}) \leftarrow \text{UniverseSetup}(U^g)$
16:    Collect from replies the maximum (limiting by at most $t+1$) set of MTS partial signatures $S^g_{J^g} = \{\text{Sign}_j(x, y)\}_{j \in J^g}$
   from $J^g \subseteq U^g$.
17:    $g \leftarrow (x, y, |J^g|, \text{Combine}(\text{CK}_{U^g}, S^g_{J^g}), indirects^g, culprit)$
18: end for
19: return groups
```

If it is achieved, p_ℓ indicates verifiable vector collection has been successfully achieved, and invites processes to participate in the persuasion protocol. At this very moment, p_ℓ (if correct) knows it will able to build and broadcast a proof-of-dispersal by the end of the persuasion protocol.

C.7.2 Security of Verifiable Vector Collection. We prove that ADA-DISP$_{\text{val}}$ satisfies val-local-accuracy as long as the sub-routine persuade$_{\text{val}}$ satisfies val-local-accuracy.

Lemma 26 (val-local-accuracy of ADA-DISP$_{\text{val}}$)**.** Let val be a validity property. If persuade$_{\text{val}}$ satisfies val-local-accuracy, then ADA-DISP$_{\text{val}}$ satisfies val-local-accuracy.

PROOF. No ACCUSE or DISCLOSE message is signed outside the sub-routines LearnOrExpose$_{\text{val}}$ and persuade$_{\text{val}}$ except at line 15 of Algorithm 11, where the entry is consistent with the starting value of Algorithm 5 (see line 16 of Algorithm 5). Sub-protocol persuade$_{\text{val}}$ is assumed to satisfy val-local-accuracy, while Lemma 15 proves that LearnOrExpose$_{\text{val}}$ satisfies val-local-accuracy as well. Here again, the entry is consistent with the starting value (see line 29 of Algorithm 11). Therefore, the lemma holds. □

We are finally able to prove Lemma 8 of accuracy.

Lemma 27 (ADA-DISP's local accuracy)**.** The following holds:

- ADA-DISP$_{\text{val}_{ic}}$ satisfies val$_{ic}$-local-accuracy.
- ADA-DISP$_{\text{val}_{sv}}$ satisfies val$_{sv}$-local-accuracy.

PROOF. val$_{ic}$-local-accuracy is ensured by Lemma 26 and Lemma 19, whereas val$_{sv}$-local-accuracy is guaranteed by Lemma 26 and Lemma 22. □

We recall that Lemma 27 (identical to Lemma 8) implies Lemma 10, i.e., the following holds:

- VerifyVector$_{\text{val}_{ic}}$ is unforgeable for (ADA-DISP$_{\text{val}_{ic}}$, val$_{ic}$).
- VerifyVector$_{\text{val}_{sv}}$ is unforgeable for (ADA-DISP$_{\text{val}_{sv}}$, val$_{sv}$).

Next, we explain how the liveness of Verifiable Vector Collection is achieved.

Algorithm 11 pod_creation_attempt$_{\text{val}}$($start\_value_i$, p_ℓ, $indirects_i$, $culprits_i$, $contact_i^{in}$, $contact_i^{out}$): Pseudocode (for process p_i)

```
1: Input Parameters:
2:     Value $start\_value_i$                                   ▷ the input value; first parameter
3:     Process $p_\ell$                                          ▷ the leader; second parameter
4:     Map(Process → Disclose_Msg) $indirects_i$                ▷ storing values indirectly obtained in former instances
5:     Map(Process → T_Signature) $culprits_i$                  ▷ storing PoM indirectly obtained in former instances
6:     Set(Process) $contact_i^{in}$      ▷ Processes that have requested the input of $p_i$ in a former LearnOrExpose instance
7:     Set(Process) $contact_i^{out}$     ▷ Processes whose input has been asked by $p_i$ in a former LearnOrExpose instance
8: Variables:
9:     Digest_Value $certified\_digest_i \leftarrow \bot$
10:    PoD $pod_i \leftarrow \bot$                                ▷ succinct proof of dispersal
11:    Map(Process → Disclose_Msg) $discloses_i \leftarrow empty$   ▷ storing input entries
12:    Entry $\triangleq$ Digest_Value $\cup$ Value $entry_i \leftarrow \bot$   ▷ the entry is either the proposal or its digest, depending on val
13: Round 1:
14:    $entry_i \leftarrow \text{entry}_{\text{val}}(start\_value_i)$   ▷ either Digest($start\_value_i$) or $start\_value_i$ depending on val
15:    send ⟨DISCLOSE, $entry_i$, $\text{ShareSign}_i^{t+1}$(DISCLOSE, $entry_i$)⟩ to $p_\ell$   ▷ inform the leader of the starting entry
16: Round 2:                                                   ▷ executed only by $p_\ell$
17:    if $p_i = p_\ell$:
18:        let $discloses_i[p_j] \leftarrow m_j \triangleq$ ⟨DISCLOSE, $e_j$, $\text{ShareSign}_j^{t+1}$(DISCLOSE, $e_j$)⟩ if $m_j$ is received
19:        for each process $p_j$, $discloses_i[p_j] \leftarrow indirects_i[p_j]$   ▷ complete with LearnOrExpose's messages
20:        if $|discloses_i.\text{Keys}() \cup culprits_i.\text{Keys}()| = 2t+1$:   ▷ check if VCC has been achieved
21:            ▷ reaching this point ensures the pre-condition of optimistic termination of the Persuasion problem, that is $\text{VerifyVector}_{\text{val}}(vec, proof)$ with $vec$ = extract_vector($discloses$, $culprits$), and $proof$ = extract_vector_proof($discloses$, $culprits$))
22:            broadcast ⟨SUCCESS⟩
23:        else:                                               ▷ Select a new process to accuse
24:            broadcast ⟨FAIL, $p_a$⟩ with $p_a \notin (discloses_i.\text{Keys}() \cup culprits_i.\text{Keys}())$
25: Round 3:
26:    if ⟨SUCCESS⟩ is received from $p_\ell$:
27:        invoke $\text{persuade}_{\text{val}}(start\_value_i, p_\ell, discloses_i, culprits_i)$
28:    else if ⟨FAIL, $p_a$⟩ is received from $p_\ell$:   ▷ collect one more disclose message or PoM for the next view
29:        invoke $\text{LearnOrExpose}_{\text{val}}(p_\ell, start\_value_i, p_a, indirects_i, culprits_i, contact_i^{in}, contact_i^{out})$
30: Round 3+$r_{\text{val}}$:   ▷ $r_{\text{val}} \in O(1)$ is the maximum number of rounds between persuasion$_{\text{val}}$ and LearnOrExpose
31:    $(certified\_digest_i, pod_i) \leftarrow$ outputs of persuasion$_{\text{val}}$ (if previously invoked)
32:    $(indirects_i, culprits_i, contact_i^{in}, contact_i^{out}) \leftarrow$ outputs of LearnOrExpose (if previously invoked)
33:    return $certified\_digest_i, pod_i, indirects_i, culprits_i, contact_i^{in}, contact_i^{out}$
```

C.7.3 Liveness of Verifiable Vector Collection. We now prove that the verifiable vector collection (VCC) is adaptively achieved.

Definition 5. [Achieving verifiable vector collection (VCC)] An iteration $\mathcal{I}$ of the protocol pod_creation_attempt$_{\text{val}}$ (Algorithm 11) invoked by ADA-DISP$_{\text{val}}$ (Algorithm 5) is said to *achieve verifiable vector collection (VCC) under* p_ℓ if persuade$_{\text{val}}(\cdot, p_\ell, \cdot, \cdot)$ is triggered in the same round by all the correct processes at line 27 of Algorithm 11 for a correct leader p_ℓ that has collected $(vec, proof)$, with $\text{VerifyVector}_{\text{val}}(vec, proof) = true$, where vec = extract_vector($discloses_\ell$, $culprits_\ell$), and $proof$ = extract_vector_proof($discloses_\ell$, $culprits_\ell$).

ADA-DISP$_{\text{val}}$ (Algorithm 5) is said to *achieve verifiable vector collection (VCC) at view* ℓ under a (correct) leader p_ℓ if the corresponding ℓ-th iteration of the protocol pod_creation_attempt$_{\text{val}}$ (Algorithm 11) called by ADA-DISP$_{\text{val}}$ achieves verifiable vector collection under (correct) leader p_ℓ.

For a persuade$_{\text{val}}$ protocol solving the Persuasion problem, its execution upon verifiable vector collection will terminate the dispersion. We aim to establish the liveness of the verifiable vector collection. The entry of process p_i is said to be *reported* if, for every correct process p_j, p_i is included in either *indirect*$_j$.Keys() or *culprits*$_j$.Keys(). Furthermore, an iteration $\mathcal{I}$ of the pod_creation_attempt subprotocol is considered *unsuccessful* if it is led by a correct leader that fails to produce a PoD by the end of the instance.

To prove liveness, we first establish that an unsuccessful iteration leads to an increase in the number of reported Byzantine entries.

Lemma 28. Consider a specific iteration $\mathcal{I}$ of the pod_creation_attempt subprotocol (Algorithm 11) with the following conditions: (1) all correct processes participate in $\mathcal{I}$, (2) the leader p_ℓ of $\mathcal{I}$ is correct, and (3) the leader does not reach line 27. Then, the count of reported Byzantine entries increments by one.

Proof. All correct processes transmit their entry values to p_ℓ (line 15). Since the leader does not reach line 27, p_ℓ accuses a process p_a (line 24) in a LearnOrExpose instance initiated at line 29. As stated in line 24, p_a's entry is not yet reported. All correct processes participate in this instance with a commonly accused process p_a. By Lemma 16, the entry of p_a becomes reported by the end of the LearnOrExpose instance. □

Second, we show that when all Byzantine entries are reported, a correct leader will necessarily be successful in its iteration.

Lemma 29. Consider a specific iteration $\mathcal{I}$ of the pod_creation_attempt sub-protocol (Algorithm 11) assuming that (1) all correct processes are participating in $\mathcal{I}$, (2) the leader p_ℓ of $\mathcal{I}$ is correct, and (3) all Byzantine entries have been reported. Then, $\mathcal{I}$ achieves verifiable vector collection (VCC) under p_ℓ.

Proof. The check at line 20 passes since all Byzantine entries have been reported, while all correct processes send their entry to p_ℓ at line 15. □

Lemma 30 (Verifiable Vector Collection)**.** If not all the correct processes acquired a pair $(d, \text{PoD}(d))$ by view $2f$, then ADA-DISP$_{\text{val}}$ achieves verifiable vector collection (VCC) at view $\ell = 2f + 1$ under (an honest leader) p_ℓ.

Proof. For each iteration overseen by a correct leader p_ℓ, if one correct process already has a PoD, all correct processes obtain a PoD in the view driven by this leader by Lemma 11.

Otherwise, every correct process initiates the pod_creation_attempt sub-protocol (line 16). If an iteration is unsuccessful, Lemma 28 guarantees that the count of reported entries increases by one. Consequently, there can be at most f unsuccessful iterations driven by an honest leader, as the complete reporting of the at most f Byzantine entries results in a successful iteration according to Lemma 29. Therefore, after a maximum of $2f$ iterations (comprising at most f iterations led by Byzantine leaders and at most f unsuccessful iterations led by correct leaders), a successful iteration under a correct leader is ensured. □

C.8 Proof of Correctness of ADA-DISP

Lemma 31. Let val be a validity property. Let persuade$_{\text{val}}$ be a protocol that solves the Persuasion problem for val and VerifyVector$_{\text{val}}$. Then, ADA-DISP$_{\text{val}}$ solves the Dispersion problem referring to the validity property val.

Proof. Observe that an ACK message can only be partially-signed in the persuade$_{\text{val}}$ sub-routine. Hence, integrity, validity, and observation are ensured by the specification of the Persuasion problem solved by persuade$_{\text{val}}$. Let us prove termination. By Lemma 30, if not all the correct processes acquired a pair $(d, \text{PoD}(d))$ by view $2f + 1$, then an iteration of pod_creation_attempt

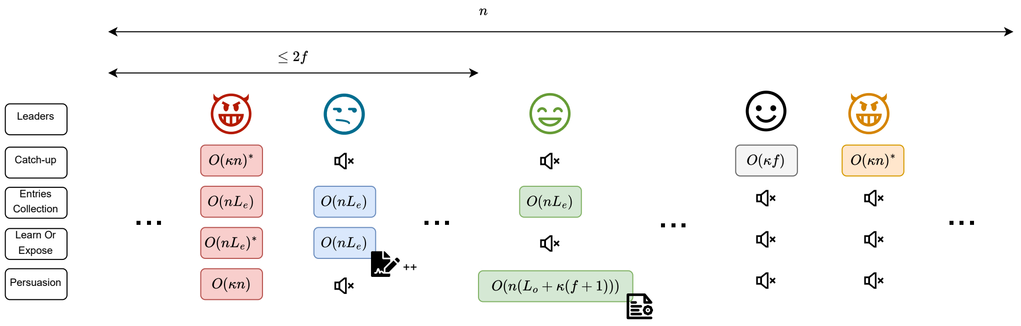


Fig. 8. Disperse. L_e denotes the maximum between κ and the size of the entry (L_{in} or κ depending on the validity property). The $O(\cdot)^*$ notation means the average over all the views driven by a Byzantine leader. The green view represents the successful leader if the delivery of the proof-of-dispersal (PoD) has not been achieved under a Byzantine leader before. There are at most $2f$ views preceding the successful one, with at most f Byzantine leaders and at most f unsuccessful correct leaders. Each unsuccessful leader drives a LearnOrExpose instance, where the number of reported entries is incremented by one. These successive LearnOrExpose instances will allow the successful leader to collect for each process either a signed entry or an undeniable proof-of-misbehaviour. This will be enough to build and broadcast a PoD in a persuasion instance. The correct processes are silent in the remaining views, except when forwarding a PoD to complainers.

achieves verifiable vector collection (VCC) under (an honest leader) $p_{\ell \leq 2f+1}$. By definition, it means $\text{persuade}_{\text{val}}(\cdot, p_\ell, \cdot, \cdot)$ is triggered in the same round by all the correct processes at line 27 for a correct leader $p_{\ell \leq 2f+1}$ that has collected $(vec, proof)$ with $\text{VerifyVector}_{\text{val}}(vec, proof) = true$, where $vec = \text{extract\_vector}(discloses_\ell, culprits_\ell)$, $proof = \text{extract\_vector\_proof}(discloses_\ell, culprits_\ell)$. By optimistic termination of the Persuasion problem solved by $\text{persuade}_{\text{val}}$, every correct process obtains a pair $(d, \text{PoD}(d))$ at line 31 of Algorithm 11, passed on by line 33 of Algorithm 11 to line 19 of Algorithm 5, and finally acquired at line 21 of Algorithm 5. Hence, termination is also satisfied, which concludes the proof. □

THEOREM 3. *Let* $\text{val}_{ic} = \text{interactive\_consistency}$. *Then* ADA-DISP$_{\text{val}_{ic}}$ *solves the Dispersion problem referring to the validity property* val_{ic}.

PROOF. By Lemma 31 and Lemma 20. □

THEOREM 4. *Let* $\text{val}_{vs} = \text{strong\&external\_validity}$. *Then* ADA-DISP$_{\text{val}_{vs}}$ *solves the Dispersion problem referring to the validity property* val_{vs}.

PROOF. By Lemma 31 and Lemma 25. □

C.9 Proof of Complexity of ADA-DISP

The communication complexity of ADA-DISP is illustrated in Figure 8. We will devote this whole subsection to formally proving the complexity.

Proof of VCC's complexity. We recall that we denote by $\mathcal{I}^*$ the first iteration such that all the correct processes output a PoD by the end of $\mathcal{I}^*$. Moreover, we recall Lemma 14 that states that the communication complexity of all the iterations in $(\mathcal{I}^* : n]$ of the for loop of ADA-DISP (Algorithm 5) is at most $O(\kappa n f)$. This is why we focus on iterations in $[1 : \mathcal{I}^*]$.

We establish that the total exchange of DISCLOSE messages is limited to $O(n(f+1))$ across all iterations.

Lemma 32. In all iterations of the pod_creation_attempt subprotocol (Algorithm 11) (initiated at line 16 of Algorithm 5), all correct processes send at most $O(n(f+1))$ DISCLOSE messages in total.

PROOF. According to Lemma 18, no process sends a DISCLOSE message to the same process more than once via the LearnOrExpose primitive. Consequently, the total number of DISCLOSE messages directed to Byzantine processes is at most $O(n(f+1))$. Moreover, a correct process issues at most one REQUEST_INPUT message in a single LearnOrExpose instance, and a DISCLOSE message is sent only in response to such a REQUEST_INPUT message. Therefore, the exchange of DISCLOSE messages between correct processes is limited to $O(n)$ in a single LearnOrExpose instance, and by extension, in a single pod_creation_attempt subprotocol instance (Algorithm 11). Given that the final iteration $\mathcal{I}^*$ is preceded by no more than $2f$ iterations (including at most f unsuccessful iterations led by correct leaders and at most f iterations led by Byzantine leaders), the total exchange of DISCLOSE messages in ADA-DISP$_{\mathsf{val}}$ (Algorithm 5) does not exceed $O(n(f+1))$. □

We now aim to demonstrate that the number of bits transmitted in a specific iteration before iteration $\mathcal{I}^*$ is limited to $O(\kappa n)$ if we focus on non-DISCLOSE messages outside the persuade$_{\mathsf{val}}$ protocol.

Lemma 33. Consider $\mathcal{I}$, an iteration of the for loop in the ADA-DISP$_{\mathsf{val}}$ protocol (Algorithm 5). The number of bits sent in $\mathcal{I}$ that do not belong to either a DISCLOSE message or a message sent in the persuade$_{\mathsf{val}}$ protocol is then $O(\kappa n)$.

PROOF. Excluding DISCLOSE messages and persuade$_{\mathsf{val}}$'s messages, the communication necessarily includes the leader. Disregarding (a) persuade$_{\mathsf{val}}$'s messages, (b) DISCLOSE messages (sent at line 17 of Algorithm 6), and (c) FAIL messages (sent at line 24 of Algorithm 11) with a process id of size $\log(n)$), each remaining message comprises at most $O(\kappa)$ bits words. These messages include:

- Messages in Algorithm 4: AID_REQ messages (line 7 or line 9), AID_REPLY messages (sent at line 12 with an attached succinct PoD) and AID_RELAY messages (sent at line 18 with an attached succinct PoD).
- Messages in Algorithm 6: REQUEST_INPUT messages (sent at line 11), ACCUSE messages (sent at line 24 with a partial signature), EXPOSE messages (sent at line 29 with a threshold signature), and LEARN messages (sent line 31 with a partial signature).
- Messages in Algorithm 11 excluding Algorithm 6, persuade$_{\mathsf{val}}$'s messages, DISCLOSE, and FAIL messages: SUCCESS messages (sent at line 22).

All these messages account for $O(n)$ messages with $O(\kappa)$ bits each in iteration $\mathcal{I}$. Furthermore, FAIL messages (broadcasted at line 24 of Algorithm 11 with a process id of size $\log(n)$) contribute $O(n\log(n))$ bits.

Finally, recall that we assume $\kappa > \log(n)$. Therefore, the total bits will be $O(\kappa n)$. □

Lemma 34. Excluding persuade$_{\mathsf{val}}$'s messages, ADA-DISP$_{\mathsf{val}}$ exchanges $O(n(f+1))$ DISCLOSE messages and $O(\kappa n(f+1))$ bits. Moreover, persuade$_{\mathsf{val}}$ is executed at most f times under a Byzantine leader and at most once under a correct leader.

PROOF. We categorize all iterations of the for loop into three distinct classes: (1) Iteration $\mathcal{I}^*$, (2) the set of iterations $\mathcal{J}^{\leq}$ preceding $\mathcal{I}^*$ (included), and (3) the set of iterations $\mathcal{J}^{>}$ following $\mathcal{I}^*$ (excluded). Observe that $|\mathcal{J}^{\leq}| \leq 2f+1$, accounting for at most f iterations led by Byzantine leaders and at most f unsuccessful iterations led by honest leaders.Obviously, $|\mathcal{J}^{>}| < n$, given there are no more than n iterations.

First, a maximum of $O(n(f+1))$ DISCLOSE messages is exchanged in total, as established by Lemma 32. We now focus on non-DISCLOSE messages and non-persuade$_{\mathsf{val}}$'s messages:

- $\mathcal{J}^{\leq}$: By Lemma 33, no more than $O(\kappa n(f+1))$ bits are exchanged in $\mathcal{J}^{\leq}$.
- $\mathcal{J}^{>}$: Following Lemma 14, a maximum of $O(n(f+1)\kappa)$ bits are exchanged for iterations succeeding $\mathcal{I}^*$.

Thus, in total, excluding persuade$_{\mathsf{val}}$'s messages, no more than $O(\kappa n(f+1))$ bits and $O(n(f+1))$ DISCLOSE messages are exchanged between correct processes in ADA-DISP$_{\mathsf{val}}$.

By Lemma 30, if not all the correct processes acquired a pair $(d, \text{PoD}(d))$ by view $2f+1$, then an iteration of pod_creation_attempt achieves verifiable vector collection (VCC) under (a correct leader) $p_{\ell \leq 2f+1}$, which means until $\mathcal{I}^*$, persuade$_{\mathsf{val}}$ is executed at most f times under a Byzantine leader. After $\mathcal{I}^*$, persuade$_{\mathsf{val}}$ is not executed by correct processes. Therefore, it will be executed at most once under a correct leader. □

Lemma 35. Let val be a validity property. Let L_e be the maximum between κ and the size of an entry (L_{in} or κ depending on val). Assume the communication complexity of persuade$_{\mathsf{val}}$ under a Byzantine leader is $O(\kappa n)$. Then, ADA-DISP$_{\mathsf{val}}$ exchanges $O(n(f+1)(L_e+\kappa))$ bits to achieve verifiable vector collection.

PROOF. Follows from Lemma 34. □

We now focus on the communication complexity of persuade$_{\mathsf{val}}$.

Proof of persuasion complexity for interactive consistency.

Lemma 36. The execution of persuade$_{\mathsf{val}_{ic}}$ (Algorithm 7) under a leader p_ℓ incurs an exchange of (1) $O(\kappa n)$ bits if p_ℓ is Byzantine, and $O(n^2(L_{in}+\kappa))$ bits otherwise.

PROOF. First, it is important to note that only a leader can send a VECTOR message with $O(n)$ entries and $O(n)$ signatures. Every non-VECTOR message represents at most $O(\kappa)$ bits, and in each invocation, a non-leader only sends $O(1)$ such messages. Thus, the lemma. □

Proof of persuasion complexity for strong validity. First, let us recall Lemma 21, which establishes that a correct process can partition up to $2t+1$ values into a maximum of $3 \in O(1)$ groups. This partitioning ensures that (1) each group contains no more than t values, and (2) any two adjacent groups collectively contain at least $t+1$ values.

A direct implication of Lemma 21 is that any bucket certificate produced by a correct leader comprises $O(\kappa(f+1))$ bits.

Lemma 37. Any bucket certificate (positive or negative) generated by a correct leader contains at most $O(\kappa(f+1))$ bits.

PROOF. A positive bucket certificate contains $O(\kappa)$ bits, as it is merely a threshold signature. On the other hand, a negative bucket certificate contains $O(\kappa + f(\kappa + \log(n)))$ bits. Recall that the negative bucket certificate comprises a set of at most $3 \in O(1)$ extended groups (by Lemma 21). Each extended group includes (a) an $O(\kappa)$-sized universe threshold signature, (b) $O(f)$ proofs-of-misbehaviour, each of size $O(\kappa)$, and (c) as many as f DISCLOSE messages from Byzantine processes along with their process id description, each of size $O(\kappa + \log n)$. Finally, we recall that it is assumed that $\kappa > \log(n)$. Therefore, a negative bucket certificate contains $O(\kappa(f+1))$ bits. □

We now establish that the communication cost associated with persuade$_{\mathsf{val}}$ is linear except for the leader that can send a certificate of $O(\kappa(f+1))$ bits.

Lemma 38. The execution of persuade$_{\mathsf{val}_{sv}}$ (Algorithm 8) under a leader p_ℓ incurs an exchange of (1) $O(\kappa n)$ bits if p_ℓ is Byzantine, and $O(nL_o + n(f+1)\kappa)$ bits otherwise.

Proof. First, it is important to note that only a leader can send a NEG_CERT message with a negative certificate Σ (as specified at line 31). In such a situation, the leader must send its input value as a default externally valid value. This implies the nL_o bits will be sent in this scenario. If the leader is correct, Σ contains at most $O(\kappa n(f+1))$ bits, by Lemma 37. Every non-NEG_CERT message represents $O(\kappa)$ bits and in each invocation, a non-leader only sends $O(1)$ such messages. Thus, the lemma. □

Proof of dispersion complexity. Finally, we are ready to demonstrate the complexity of ADA-DISP$_{\mathsf{val}}$.

Theorem 5. *ADA-DISP$_{\mathsf{val}_{ic}}$ exchanges $O(n^2(L_{in}+\kappa))$ bits.*

Proof. Follows from lemmas 35 and 36. □

Theorem 6. *ADA-DISP$_{\mathsf{val}_{sv}}$ exchanges $O(nL_o + n(f+1)\kappa)$ bits.*

Proof. Follows from lemmas 35 and 38. □

C.10 ADA-DISP$_{\mathsf{val}_{ic}}$ with $O(n^2L_{in}+\kappa fn)$ bit-complexity instead of $O(n^2L_{in}+\kappa n^2)$

In this section, we explain how to achieve Dispersion with $O(n^2L_{in}+\kappa fn)$ bit-complexity instead of $O(n^2L_{in}+\kappa n^2)$ bit-complexity, via a non-interactive n-aggregate signature scheme. Such a scheme lets each of the n processes sign a different message, but all these signatures can be aggregated into a single short signature of $O(\kappa)$ bits. This short signature can convince the verifier (equipped with a description of the signers) that all signers signed their designated message.

Aggregate signature scheme. A *non-interactive n-aggregate signature scheme* is a tuple of efficient (local) protocols $(\mathsf{Keys}, \mathsf{ShareSign}, \mathsf{ShareVerify}, \mathsf{Verify}, \mathsf{Aggregate})$, where $\mathsf{Keys} = \big(\mathsf{SK} = (sk_1, ..., sk_n), \mathsf{VK} = (vk_1, ..., vk_n)\big)$, with:

- VK: a vector of verification keys stored by every correct process.
- SK: a vector of private key shares such that, for every correct process p_i, p_i stores its private key share sk_i; sk_i is hidden from the adversary.
- $\mathsf{ShareSign}(m, sk_i)$: a (potentially probabilistic) algorithm that takes (1) a string $m \in \mathsf{String}$, and (2) a private key share sk_i as the input. The algorithm outputs a signature share σ_i^S of (at most) κ bits.
- $\mathsf{ShareVerify}(m, vk_j, \sigma_j^S)$: a deterministic algorithm that takes (1) a string $m \in \mathsf{String}$, (2) the verification key vk_j of a process p_j, and (3) a signature share σ_j^S as the input. The algorithm outputs $\top$ or $\bot$ depending on whether σ_j^S is deemed as a valid signature.
- $\mathsf{Aggregate}(\{(m_i, \sigma_i)\}_{i\in S \wedge S\subseteq[1,n]}\})$: an algorithm that takes (1) a subset S of any size of message-signature share pairs. The algorithm outputs an aggregate signature σ^A.
- $\mathsf{Verify}(\{m_i\}_{i\in S\wedge S\subseteq[1,n]}, \sigma^A, B)$: a deterministic algorithm that takes (1) an ordered subset S of any size of strings $\{m_i\}_{i\in S\wedge S\subseteq[1,n]}$, (2) an aggregate signature σ^A, and (3) a bit-mask $B \in \{0,1\}^n$. The algorithm outputs $\top$ or $\bot$ depending on whether σ^A is deemed as a valid aggregate signature with reference to B.

The following properties hold:

- *Correctness of signature shares:* For every $i \in [1,n]$, $\mathsf{ShareVerify}\big(m, vk_i, \mathsf{ShareSign}(m, sk_i)\big)$ returns $\top$.
- *Unforgeability of signature shares:* If $\mathsf{ShareVerify}(m, vk_j, \sigma_j^S)$ returns $\top$, then (1) $\sigma_j^S \leftarrow \mathsf{ShareSign}(m, sk_j)$ has been executed by p_j, or (2) p_j is faulty.
- *Correctness of aggregate signatures:* Consider any (ordered) set of strings $\{m_i\}_{i\in S\wedge S\subseteq[1,n]} \subset \mathsf{String}$ and any bit-mask $B \in \{0,1\}^n$. The following holds: $\mathsf{Verify}\big(\{m_i\}_{i\in S\wedge S\subseteq[1,n]}, \mathsf{Aggregate}(\{\mathsf{ShareSign}(m_j, sk_j)\}_{B[j]=1}), B\big)$ returns $\top$.

- *Unforgeability of aggregate signatures:* If $\mathsf{Verify}(\{m_i\}_{i \in S \wedge S \subseteq [1,n]}, \sigma^A, B)$ returns $\top$, then, for every $j \in [1, n]$ such that $B[j] = 1$, (1) $\sigma_j^S \leftarrow \mathsf{ShareSign}(m_j, sk_j)$ has been executed by p_j, or (2) p_j is faulty.

There exist transparent aggregate signature schemes such that the aggregate signatures have a size of $O(\kappa)$ bits (e.g., [12]). The interface of an aggregate signature scheme is similar to the interface of the non-transparent threshold signature schemes. We emphasize that an aggregate signature σ^A has to be associated with a bit-mask B of n bits (representing the subset of signers). Indeed, this bit-mask is an argument of the associated Verify protocol. Thus, proving that a group of a linear number of processes have signed a certain set of messages requires $O(\kappa + n)$ bits.

Also, the setup is transparent. Indeed, the secret keys can be generated independently, and the correct processes have to agree on the associated verification keys (exactly as in a PKI).

A minor modification to ADA-DISP$_{\mathsf{val}_{ic}}$. Now we explain how to modify ADA-DISP$_{\mathsf{val}_{ic}}$ to achieve $O(n^2 L_{in} + \kappa f n)$ bit-complexity instead of $O(n^2 L_{in} + \kappa n^2)$ bit-complexity.

We apply the following modifications:

- Each DISCLOSE message is backed by a corresponding aggregate signature share.
- The leader of $\mathsf{persuade}_{\mathsf{val}_{ic}}$ (Algorithm 8) broadcasts (1) vec, (2) up to f PoMs of size κ, and (3) an aggregate signature of the remaining DISCLOSE messages in vec (instead of up to $n - f$ signature shares).

This yields a version of ADA-DISP$_{\mathsf{val}_{ic}}$ achieves $O(nL_o + \kappa n(f + 1))$ bit-complexity with $L_o = nL_{in}$, and so bit-optimality if $L_{in} = \Omega(\kappa f / n)$ instead of $L_{in} = \Omega(\kappa)$.

THEOREM 7. *ADA-DISP$_{\mathsf{val}_{ic}}$ with additional aggregate signatures exchanges $O(n^2 L_{in} + \kappa f n)$ bits.*

PROOF. The proof follows that of Theorem 5, where the bit complexity of $\mathsf{persuade}_{\mathsf{val}_{ic}}$ under a correct process is $O(\kappa(f + 1)n)$. □

D ADAPTIVE VALIDATED BYZANTINE AGREEMENT

In this section, we present a modification to **CKS**[19] [32] so it satisfies external validity. Therefore, making **CKS** a VBA protocol. We first explain the outline of **CKS** in Appendix D.1. We then explain the reason why **CKS** does not satisfy external validity in Appendix D.2. Based on that, we propose some modifications to **CKS** to make it satisfy external validity in Appendix D.3. Finally, we prove the correctness of the modification and the complexity of **CKS** after applying the modifications in Appendix D.4.

D.1 Outline of CKS

Essentially, **CKS** adheres to the silent views paradigm (see §2 and [71]) and leverages the key observation that setting an optimistic threshold t_o at $n - \lceil \frac{n+t+1}{2} \rceil$ enables quorum intersection even if $n = 2t + 1$. Classical "locking" mechanism guaranteeing safety across multiple views, as employed in [16, 18, 75], can be derived from the quorum intersection. When $f \leq t_o$, the first correct leader ensures that all correct parties agree on an admissible value (if it is not already the case). Subsequent honest leaders refrain from "wasting" communication in their views, thereby remaining "silent". This results in f views to reach a correct leader, after which no further communication occurs between correct processes, leading to an adaptive $(\kappa n(f + 1))$ bit complexity for $O(\kappa)$-bits values. In cases where $f > t_o$, a fallback mechanism is activated to achieve agreement using the optimally-resilient protocol proposed by Momose and Ren (which we refer to as **MR**) with $O(\kappa n^2)$ bit complexity for $O(\kappa)$-bits values [61]. We note that the adaptiveness is maintained by the fallback protocol as $f \in \Omega(n)$ whenever the fallback protocol is employed.

[19]**CKS** incorporates the correction of a minor technical flaw in [32] addressed by [38].

D.2 Why Does CKS Not Satisfy External Validity?

By itself, **CKS** does not satisfy external validity. To be precise, **CKS** has the following guarantees:

- If $f \leq t_o$, all correct processes decide a valid (satisfying $\mathsf{valid}_{\mathbf{CKS}}(\cdot)$) value v.
- If $f > t_o$, all correct process either decide a valid value v or $\perp$. If $\perp$ is decided, it must be the case that correct processes have started with different inputs.

When $\perp$ is decided, it must be because **CKS** is unable to decide in its optimistic path due to $f > t_o$. In this case, **CKS** employs **MR** as its fallback protocol. **MR** itself is a Byzantine agreement protocol tolerating $t < n/2$ faulty processes. **MR** provides strong validity, but not external validity. Namely, it is possible for **MR** to decide any value if correct processes start with different inputs. In **CKS**, when **MR** is executed, it has to check the output of **MR**. If the output is a valid value then it will be the decision, otherwise $\perp$ will be the decision. Meanwhile, $\perp$ does not necessarily satisfy external validity.

D.3 Obtaining External Validity with CKS

Here, we propose some modifications to **MR** so it satisfies external validity. By plugging this modified **MR** to **CKS**, we obtain a Byzantine Agreement protocol satisfying external validity.

We first describe how **MR** works. In a nutshell, **MR** is a recursive algorithm. **MR** partition the processes into two halves with roughly equal size, where each half runs **MR** (among themselves). When the number of processes is small (i.e., constant) , **MR** executes any Byzantine Agreement protocol satisfying strong validity, whose detail is left as a black box. Finally, when a recursion unrolls, there are some technicalities to merge the result of **MR** from the two partitions. For interested readers, we defer all the details of **MR** to [61].

We now provide the modifications to make **MR** satisfy external validity.

(1) Correct processes ignore any messages containing an invalid value.
(2) Set the constant threshold for recursion to be 1. Then, in the base case of the recursion, a process simply outputs its input value as the decision.

For simplicity, let us denote the **MR** protocol after applying the modifications as "modified **MR**".

D.4 Analysis

We now prove that modified **MR** is a BA protocol that satisfies both strong validity and external validity. Moreover, similarly to **MR**, modified **MR** terminates in $O(n)$ rounds and exchanges $O(n^2(L_o + \kappa))$ bits.

Lemma 39. The modified **MR** is a BA protocol that satisfies both strong and external validity.

Proof. If there is only 1 process, all properties are trivially satisfied. Otherwise, observe that, under the first modification, a correct process will only send messages containing a valid value. Hence, each execution $\mathcal{E}$ in the modified **MR** is equivalent to an execution $\mathcal{E}'$ in **MR** where each message containing an invalid value (which must be sent by a faulty process) is omitted instead. Therefore, termination, agreement, and strong validity are satisfied. Finally, each correct process only decides a valid value, so external validity is also satisfied. □

Lemma 40. The round complexity of the modified **MR** is $O(n)$, and its bit complexity is $O(n^2(L_o + \kappa))$.

Proof. The modifications do not add nor remove any rounds or messages from the correct processes. Hence, the complexities of the modified **MR** follow the complexities of **MR** which are: (1) $O(n)$ round complexity, and (2) $O(n^2(L_o + \kappa))$ bit complexity. □

Finally, by replacing the fallback protocol employed by **CKS** with the modified **MR**, we obtain a BA protocol satisfying external validity (i.e., we obtain a VBA protocol). Moreover, the protocol terminates in $O(n)$ rounds and exchanges $O(n(f+1)(L_o+\kappa))$ bits.

Lemma 41 (Adaptive VBA exists)**.** There exists a VBA protocol that terminates in $O(n)$ rounds and exchanges $O(n(f+1)(L_o+\kappa))$ bits.

E ADA-RETRIEVE

In this section, we present the retrieval problem along with the complete implementation and proofs of ADA-RETRIEVE, our implementation for the aforementioned problem. We first describe the retrieval problem along with our implementation in Appendix E.1. We then prove its correctness and complexity in Appendix E.2.

E.1 The Retrieval Problem

In this subsection, we recall the specification of the retrieval problem (Appendix E.1.1) and give an overview of ADA-RETRIEVE, our implementation of the problem (Appendix E.1.2).

E.1.1 Specification. In a nutshell, the goal of the retrieval problem is to ensure that, given a common digest where some correct processes have the pre-image of the digest, all correct processes output (retrieve) the pre-image. Formally, the retrieval problem exposes the following interface:

- **request** $\mathsf{input}(d \in \mathsf{Digest\_Value}, v \in \mathsf{Value} \cup \{\bot\})$: a process inputs a digest d and either $\bot$ or a value v such that $\mathsf{Digest}(v) = d$; each correct process invokes $\mathsf{input}(\cdot)$ exactly once.[20] Moreover, the following is assumed:
  - No two correct processes invoke $\mathsf{input}(d_1, v_1)$ and $\mathsf{input}(d_2, v_2)$ with $d_1 \neq d_2$.
  - At least $t+1-f$ correct processes invoke $\mathsf{input}(d, v)$ with $v \neq \bot$ (i.e., $\mathsf{Digest}(v) = d$).
- **indication** $\mathsf{output}(v' \in \mathsf{Value})$: a process outputs a value v'.

The following properties are ensured:

- *Agreement:* No two correct processes output different values.
- *Validity:* Let a correct process input a value v. No correct process outputs a value $v' \neq v$.
- *Termination:* Every correct process eventually outputs a value.

E.1.2 Implementation. Here, we will describe the implementation of ADA-RETRIEVE (see Algorithm 12). At a high level, ADA-RETRIEVE consists of two phases: 1) *to almost everywhere* phase that ensures all but $O(f)$ correct processes receive the pre-image, and 2) *to everywhere* phase that ensures all correct process receive the pre-image value.

The first phase consists of gossiping the pre-image value through an expander graph. In short, an expander graph is a graph (of a low degree, typically) with a good expansion property. Roughly speaking, it means each vertex has a few neighbours, and any "relatively small" set of vertices has many neighbours. In ADA-RETRIEVE, we use a specific expander graph from [73], which has some nice additional properties: after $O(\log n)$ rounds of gossip, all but $O(f)$ correct processes must have received the pre-image. Importantly, when a correct process received a value v' from the gossip, it can verify whether the v' is indeed the pre-image by comparing $\mathsf{Digest}(v')$ with the the common digest. Next, the second phase ensures that those remaining correct processes that have not obtained any pre-image in the first phase eventually do obtain the pre-image. This is done in two steps: (1) we ensure that each correct process p_i knows the i-th RS symbol of the pre-image, and (2) we ensure that any correct process that is left behind in the first phase obtains $t+1$ different RS symbols. Importantly, each RS symbol is accompanied by an accumulator witness, allowing a correct process to verify its validity (i.e., that the RS symbol is correctly-encoded).

[20]We underline that $d \neq \bot$ even if $v = \bot$.

ADA-RETRIEVE's description. We present the algorithm in Algorithm 12. Immediately, each process starts the *"to almost everywhere"* phase by gossipping via the expander graph for $O(\log n)$ rounds (line 9). Notably, unlike the standard notion of gossip, here a correct process only forwards a value to its neighbours at most once (line 10). Upon receiving the pre-image value during the gossip, a correct process stores it in v_i (line 11).

Upon finishing the *to almost everywhere phase*, each process continues to the *to everywhere phase*. The phase unfolds in several rounds. In the first round, a correct process p_i checks whether it has obtained the pre-image ($v_i \neq \bot$). If yes, it computes the i-th RS symbol and the witness, then stores them in $symbol_i$ and $witness_i$ (line 13). Otherwise, p_i sends a broadcast as an attempt to obtain them (line 12). In the second round, suppose a correct process p_j already has the pre-image. It then responds to each message sent at the first round, in which a message sent by a process p_i will be replied with the i-th RS symbol and its witness (line 16). In the third round, upon receiving the reply from p_j, p_i then updates $symbol_i$ and $witness_i$ accordingly (line 18). Then, p_i sends a broadcast to obtain at least $t + 1$ RS symbols, which will allow it to recover the pre-image. Observe that at this point, each correct process p_j must have obtained the j-th RS symbol and its witness. Therefore, at the fourth round, each correct process p_j will be able to reply the requests sent at the third round (line 20). In the final, fifth round, there are two cases. If a correct process p_i starts *to everywhere* phase with v_i, it can simply trigger output(v_i) (line 21). Otherwise, p_i must have collected $t + 1$ RS symbols, allowing it to recover the pre-image and thus, output the recovered value (line 25).

Algorithm 12 ADA-RETRIEVE: Pseudocode for process p_i

```
1:  Uses:
2:      ExpanderGraph, instance $G$                                              ▷ see [73]
3:  Input parameters:
4:      Digest_Value $digest_i \leftarrow d$                                     ▷ the input $d$
5:      Value $v_i \leftarrow v$                       ▷ the input $v$. if $v \neq \bot$, then $d$ = Digest$(v)$
6:  Local variables:
7:      RS_Symbol $symbol_i \leftarrow \bot$                   ▷ the $i$-th RS symbol from the pre-image
8:      Witness $witness_i \leftarrow \bot$                                ▷ witness for $symbol_i$
9:  for Integer $k \leftarrow 1$ to $O(\log n)$:                              ▷ to almost everywhere
10:     if $v_i \neq \bot$ and has not gossiped: multicast ⟨GOSSIP, $v_i$⟩ to $i$'s neighbour in $G$
11:     if received ⟨GOSSIP, Value $v'$⟩, Digest$(v') = digest_i$, and $v_i = \bot$: $v_i \leftarrow v'$
12: if $v_i = \bot$: broadcast ⟨COMPLAIN⟩                                      ▷ to everywhere
13: else: $symbol_i \leftarrow P_{v_i}(i)$; $witness_i \leftarrow$ CreateWit$(ak, digest_i, (i, symbol_i))$
14: if $v_i \neq \bot$, upon receiving ⟨COMPLAIN⟩ from $p_j$:
15:     let $s_j \leftarrow P_{v_i}(j)$ and $w_j \leftarrow$ CreateWit$(ak, digest_i, (j, s_j))$
16:     send ⟨YOUR_SYMBOL, $s_j$, $w_j$⟩ to $p_j$
17: if $v_i = \bot$, upon receiving ⟨YOUR_SYMBOL, RS_Symbol $s$, Witness $w$⟩ and Verify$(ak, digest_i, w, (i, s)) = true$:
18:     $symbol_i \leftarrow s$; $witness_i \leftarrow w$
19:     broadcast ⟨REQ_SYMBOLS⟩
20: upon receiving ⟨REQ_SYMBOLS⟩ from $p_j$: send ⟨MY_SYMBOL, $symbol_i$, $witness_i$⟩ to $p_j$
21: if $v_i \neq \bot$: trigger output$(v_i)$
22: else, upon receiving ⟨MY_SYMBOL, RS_Symbol $s_j$, Witness $w_j$⟩ from $t + 1$ different $p_j$ with
23: Verify$(ak, digest_i, w_j, (j, s_j)) = true$:
24:     reconstruct $v$ from $t + 1$ RS via decode$(\cdot)$
25:     trigger output$(v)$
```

E.2 Proof of Correctness & Complexity

Proof of correctness. We first prove that ADA-RETRIEVE satisfies the specification stated in Appendix E.1.1. We start with a lemma related to the *"to almost everywhere"* phase.

Lemma 42. At the end of the *"to almost everywhere"* phase (after the loop at line 9 terminates), either $v_i = \bot$ or $\text{Digest}(v_i) = \mathit{digest}_i$.

Proof. First, v_i is initialized as $\bot$ or such that $\text{Digest}(v_i) = \mathit{digest}_i$. Then, within the *"to almost everywhere"* phase, it can only be updated at line 11. Here, it can only be updated into a value v' such that $\text{Digest}(v') = \mathit{digest}_i$. □

We continue with several lemmas related to the *"to everywhere"* phase.

Lemma 43. At least $t + 1 - f$ correct process starts the *"to everywhere"* phase with $v_i \neq \bot$.

Proof. Follows from the pre-condition and Lemma 42 which implies a correct process cannot update v_i into $\bot$. □

Lemma 44. By the end of round 2 of the *"to everywhere"* phase (line 16), each correct process p_i updates its symbol_i to be the i-th RS symbol of $\text{Digest}^{-1}(\mathit{digest}_i)$ and $\mathit{witness}_i$ to be the witness for symbol_i.

Proof. When a correct process p_i starts the *"to everywhere"* phase, if $v_i \neq \bot$, symbol_i and $\mathit{witness}_i$ will be updated at line 13 into the i-th RS symbol of $\text{Digest}^{-1}(\mathit{digest}_i)$ and its witness, respectively. Otherwise, p_i will request for the i-th RS symbol (line 12). As at least $t - f + 1 > 0$ correct processes start the *"to everywhere"* phase with $v_i \neq \bot$ (Lemma 43), some correct process enters line 16. There, the correct process replies with the i-th RS symbol along with a witness that testifies that the reply is indeed the i-th RS symbol. Finally, upon receiving an RS symbol and a witness testifying that it is indeed the i-th RS symbol, p_i updates its symbol_i and $\mathit{witness}_i$ with the received RS symbol and witness (line 18). □

Lemma 45. At the end of the *"to everywhere"* phase, each correct process p_i which starts this phase with $v_i = \bot$ will be able to collect $t + 1$ different RS symbols of $\text{Digest}^{-1}(\mathit{digest}_i)$.

Proof. At the third round, p_i will send a request for all correct RS symbols. By Lemma 44, by the end of the second round, each correct process p_j has updated its symbol_j to be the j-th RS symbol of $\text{Digest}^{-1}(\mathit{digest}_i)$. Therefore, each correct p_j will send a reply to p_i at the fourth round, containing the j-th RS symbol. Hence, at least $n - f \geq t + 1$ different RS symbols will be collected. □

We are now ready to prove the correctness of ADA-RETRIEVE.

Lemma 46. ADA-RETRIEVE satisfies validity.

Proof. If $\text{output}(\cdot)$ is triggered at line 21, validity follows from the pre-condition and Lemma 42. Otherwise, $\text{output}(\cdot)$ will be triggered at line 25, in which validity follows from Lemma 45. □

Lemma 47. ADA-RETRIEVE satisfies agreement.

Proof. Follows from Lemma 46. As at least $t - f + 1 > 0$ correct processes input a value v, all correct processes will output v. □

Lemma 48. ADA-RETRIEVE satisfies termination.

Proof. There are finitely many rounds in ADA-RETRIEVE, and all correct processes eventually trigger $\text{output}(\cdot)$. □

As ADA-RETRIEVE satisfies validity (Lemma 46), agreement (Lemma 47), and termination (Lemma 48), ADA-RETRIEVE is correct.

Theorem 8. *ADA-RETRIEVE is correct.*

Proof of complexity. We first focus on the complexity of the *"to almost everywhere"* phase. We begin by restating the properties of the expander graph from [73].

Lemma 49. (Restated from [73]) For all n, there exists a constant $d > 0$ that is independent of n and some d-regular graph with n vertices, where $\forall T \subset V$ with $|T| \leq n/72$, there is a connected component $P(T) \in V \setminus T$ of size at least $n - 6|T|$. Furthermore, the diameter of $P(T)$ is $O(\log n)$.

The lemma above allows us to upper-bound the number of correct processes that exit the *"to almost everywhere"* phase with $output_i \neq \bot$.

Lemma 50 (All but $O(f)$). All but $O(f)$ correct processes will set their $v_i \neq \bot$ by the end of the *"to almost everywhere"* phase.

Proof. Suppose $f \leq n/72$. Observe that Lemma 49 states that, if $f \leq n/72$, there exists a connected component in the graph G excluding the faulty processes which contains at least $n - 6f$ correct processes. Moreover, this connected component has a diameter of $O(\log n)$. Recall that correct processes gossip for $O(\log n)$ rounds (line 9). If there is at least one correct process in that connected component which starts with $v \neq \bot$, then all correct processes in this component will receive v by the end of the *to almost everywhere* phase. Hence, all but $O(f)$ correct processes will receive v and update their $v_i \leftarrow v$. This is ensured if at least $6f + 1$ correct processes start with $v \neq \bot$. As $f \leq n/72$, $t - f + 1 \geq 6f + 1$ and therefore, all but $O(f)$ correct processes will set their $v_i \neq \bot$. Now, let us consider $f > n/72$. In this case, up to $O(n)$ correct processes may still have $v_i = \bot$ by the end of the *to almost everywhere* phase. However, as $f \in \Omega(n)$, it can also be treated as all but $O(f)$ correct processes have set their $v_i \neq \bot$. Therefore, in all cases, the lemma statement holds. □

We now prove the communication of each phase.

Lemma 51. The *"to almost everywhere"* phase exchanges $O(nL_o + \kappa n)$ bits.

Proof. A correct process can only send messages to its neighbours in the expander graph at line 10, notably one message with $O(L_o + \kappa)$ bits per neighbour. As the graph has a constant degree, p_i will only send messages to a constant number of processes. Moreover, p_i will only send a message to its neighbours at most once. Summing from all processes, we get that $O(nL_o + \kappa n)$ bits are exchanged. □

Lemma 52. The *"to everywhere"* phase exchanges $O(fL_o + \kappa nf)$ bits.

Proof. By Lemma 50, $O(f)$ correct processes will start the *"to everywhere"* phase with $v_i = \bot$. Next, recall that the size of an RS symbol will be $O(\max(\frac{L_o}{t+1}, \log n))$ and the size of a witness is $O(\kappa)$. We break down on each line where a correct process may send some messages:

- Line 12. This line will be executed by $O(f)$ correct processes. As the message size is $O(\kappa)$ bits, the bits exchanged are $O(\kappa nf)$.
- Line 16. Up to $O(n)$ correct process p_i with $v_i \neq \bot$ must reply to $O(f)$ processes (including faulty processes). As the message size is $O(\max(\frac{L_o}{t+1}, \log n) + \kappa)$, the bits exchanged are $O(fn \cdot (\max(\frac{L_o}{t+1}, \log n) + \kappa)) = O(fL_o + \kappa nf)$.
- Line 19. As there are $O(f)$ correct processes which will enter this line and the message size is $O(\kappa)$ bits, the bits exchanged are $O(\kappa nf)$.
- Line 20. Each correct process p_i has to send the i-th RS symbol and its witness to $O(f)$ processes (including faulty processes). As the message size is $O(\max(\frac{L_o}{t+1}, \log n) + \kappa)$, the bits exchanged are $O(fn \cdot (\max(\frac{L_o}{t+1}, \log n) + \kappa)) = O(fL_o + \kappa nf)$.

Summing everything together, we get that $O(fL_o + \kappa nf)$ bits are exchanged. □

Finally, we are now ready to prove the complexity of the ADA-RETRIEVE.

THEOREM 9. *ADA-RETRIEVE exchanges* $O(nL_o + n(f+1)\kappa)$ *bits and terminates in* $O(\log n)$ *rounds.*

PROOF. As ADA-RETRIEVE is a sequential composition of the *"to almost everywhere"* and *"to everywhere"* phases, its communication complexity (resp., latency) is the sum of the bits exchanged in (resp., latency of) each phase. The number of exchanged bits follows from Lemma 51 and Lemma 52. The latency follows from the fact that the *"to almost everywhere"* phase consists of $O(\log n)$ rounds and the *"to everywhere"* phase consists of $O(1)$ rounds. □