Every Bit Counts in Consensus

Pierre Civit
Sorbonne University, Paris, France

Seth Gilbert
National University of Singapore, Singapore

Rachid Guerraoui
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Jovan Komatovic
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Matteo Monti
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Manuel Vidigueira
Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Abstract

Consensus enables n processes to agree on a common valid L-bit value, despite $t < n/3$ processes being faulty and acting arbitrarily. A long line of work has been dedicated to improving the worst-case communication complexity of consensus in partial synchrony. This has recently culminated in the worst-case *word* complexity of $O(n^2)$. However, the worst-case *bit* complexity of the best solution is still $O(n^2L + n^2\kappa)$ (where κ is the security parameter), far from the $\Omega(nL + n^2)$ lower bound. The gap is significant given the practical use of consensus primitives, where values typically consist of batches of large size ($L > n$).

This paper shows how to narrow the aforementioned gap. Namely, we present a new algorithm, DARE (Disperse, Agree, REtrieve), that improves upon the $O(n^2L)$ term via a novel dispersal primitive. DARE achieves $O(n^{1.5}L + n^{2.5}\kappa)$ bit complexity, an effective $\sqrt{n}$-factor improvement over the state-of-the-art (when $L > n\kappa$). Moreover, we show that employing heavier cryptographic primitives, namely STARK proofs, allows us to devise DARE-Stark, a version of DARE which achieves the near-optimal bit complexity of $O(nL + n^2 poly(\kappa))$. Both DARE and DARE-Stark achieve optimal $O(n)$ worst-case latency.

2012 ACM Subject Classification Theory of computation → Distributed algorithms

Keywords and phrases Byzantine consensus, Bit complexity, Latency

Digital Object Identifier 10.4230/LIPIcs.DISC.2023.13

Related Version The full version of this paper, which includes detailed proofs, is available online [37].
Full Version: https://arxiv.org/abs/2306.00431

Funding *Seth Gilbert*: Supported in part by the Singapore MOE Tier 2 grant MOE-T2EP20122-0014.
Manuel Vidigueira: Supported in part by the FNS (#200021_215383).

1 Introduction

Byzantine consensus [65] is a fundamental primitive in distributed computing. It has recently risen to prominence due to its use in blockchains [70, 22, 4, 32, 5, 41, 39] and various forms of state machine replication (SMR) [8, 29, 64, 1, 11, 63, 87, 71, 74]. At the same time, the performance of these applications has become directly tied to the performance of consensus and its efficient use of network resources. Specifically, the key limitation on blockchain transaction rates today is network throughput [42, 86, 27]. This has sparked a large demand for research into Byzantine consensus algorithms with better communication complexity guarantees.

Consensus operates among n processes: each process proposes its value, and all processes eventually agree on a common valid L-bit decision. A process can either be correct or faulty: correct processes follow the prescribed protocol, while faulty processes (up to $t < n/3$) can behave arbitrarily. Consensus satisfies the following properties:

- *Agreement:* No two correct processes decide different values.
- *Termination:* All correct processes eventually decide.
- *(External) Validity:* If a correct process decides a value v, then $\mathsf{valid}(v) = \mathit{true}$.

Here, $\mathsf{valid}(\cdot)$ is any predefined logical predicate that indicates whether or not a value is valid.[1]

This paper focuses on improving the worst-case bit complexity of deterministic Byzantine consensus in standard partial synchrony [52]. The worst-case lower bound is $\Omega(nL + n^2)$ exchanged bits. This considers all bits sent by correct processes from the moment the network becomes synchronous, i.e., GST (the number of messages sent by correct processes before GST is unbounded due to asynchrony [85]). The nL term comes from the fact that all n processes must receive the decided value at least once, while the n^2 term is implied by the seminal Dolev-Reischuk lower bound [48, 85] on the number of messages. Recently, a long line of work has culminated in Byzantine consensus algorithms which achieve optimal $O(n^2)$ worst-case *word* complexity, where a word is any constant number of values, signatures or hashes [36, 66]. However, to the best of our knowledge, no existing algorithm beats the $O(n^2L + n^2\kappa)$ bound on the worst-case bit complexity, where κ denotes the security parameter (e.g., the number of bits per hash or signature). The n^2L term presents a linear gap with respect to the lower bound.

Does this gap matter? In practice, yes. In many cases, consensus protocols are used to agree on a large batch of inputs [77, 88, 27, 42, 86]. For example, a block in a blockchain amalgamates many transactions. Alternatively, imagine that n parties each propose a value, and the protocol agrees on a set of these values. (This is often known as vector consensus [14, 15, 50, 80, 49, 40].) Typically, the hope is that by batching values/transactions, we can improve the total throughput of the system. Unfortunately, with current consensus protocols, larger batches do not necessarily yield better performance when applied directly [46]. This does not mean that batches are necessarily ineffective. In fact, a recent line of work has achieved significant practical improvements to consensus throughput by entirely focusing on the efficient dissemination of large batches (i.e., large values), so-called “mempool” protocols [42, 86, 27]. While these solutions work only optimistically (they perform well in periods of synchrony and without faults), they show that a holistic focus on *bandwidth* usage is fundamental (i.e., bit complexity, and not just word complexity).

1.1 Contributions

We introduce DARE (Disperse, Agree, REtrieve), a new Byzantine consensus algorithm for partial synchrony with worst-case $O(n^{1.5}L + n^{2.5}\kappa)$ bit complexity and optimal worst-case $O(n)$ latency. Moreover, by enriching DARE with heavier cryptographic primitives, namely

[1] For traditional notions of validity, admissible values depend on the proposals of correct processes, e.g., if all correct processes start with value v, then v is the only admissible decision. In this paper, we focus on external validity [25], with the observation that any other validity condition can be achieved by reduction (as shown in [38]).

STARK proofs, we close the gap near-optimally using only $O(nL + n^2 poly(\kappa))$ bits. Notice that, if you think of L as a batch of n transactions of size s, the average communication cost of agreeing on a single transaction is only $\tilde{O}(ns)$ bits – the same as a best-effort (unsafe) broadcast [24] of that transaction!

To the best of our knowledge, DARE is the first partially synchronous algorithm to achieve $o(n^2L)$ bit complexity and $O(n)$ latency. The main idea behind DARE is to separate the problem of agreeing from the problem of retrieving an agreed-upon value (see §1.2 for more details). Figure 1 places DARE in the context of efficient consensus algorithms.

Protocol	Model	Cryptography	Complexity	
			$\mathbb{E}$[Bits]†	$\mathbb{E}$[Latency]†
ABC [25]‡	Async	PKI, TS [68]	$O(n^2L + n^2\kappa + n^3)$	$O(1)$
VABA [6]	Async	above	$O(n^2L + n^2\kappa)$	$O(1)$
Dumbo-MVBA [69]	Async	above + ECC [19]	$O(nL + n^2\kappa)$	$O(1)$
			Bits	Latency
PBFT [30, 17]	PSync	PKI	$O(n^2L + n^4\kappa)$	$O(n)$
HotStuff [89]	PSync	above + TS	$O(n^2L + n^3\kappa)$	$O(n)$
Quad [36, 66]	PSync	above	$O(n^2L + n^2\kappa)$	$O(n)$
DARE	PSync	above + ECC	$O(n^{1.5}L + n^{2.5}\kappa)$	$O(n)$
DARE-Stark	PSync	above + STARK	$O(nL + n^2\kappa)$	$O(n)$

Figure 1 Performance of various consensus algorithms with L-bit values and κ-bit security parameter.

† For asynchronous algorithms, we show the complexity in expectation instead of the worst-case (which is unbounded for deterministic safety guarantees due to the FLP impossibility result [54]).

‡ Threshold Signatures (TS) are used to directly improve the original algorithm.

1.2 Technical Overview

The “curse” of GST. To understand the problem that DARE solves, we must first understand why existing algorithms suffer from an $O(n^2L)$ term. “Leader-based” algorithms (such as the state-of-the-art [75, 36, 66]) solve consensus by organizing processes into a rotating sequence of *views*, each with a different leader. A view’s leader broadcasts its value v and drives other processes to decide it. If all correct processes are timely and the leader is correct, v is decided.

The main issue is that, if synchrony is only guaranteed *eventually* (partial synchrony [52]), a view might fail to reach agreement even if its leader is correct: the leader could just be slow (i.e., not yet synchronous). The inability to distinguish the two scenarios forces protocols to change views even if the current leader is merely “suspected” of being faulty. Since there can be up to t faulty leaders, there must be at least $t + 1$ different views. However, this comes at the risk of sending unnecessary messages if the suspicion proves false, which is what happens in the worst case.

Suppose that, before GST (i.e., the point in time the system becomes synchronous), the first t leaders are correct, but “go to sleep” (slow down) immediately before broadcasting their values, and receive no more messages until GST + δ due to asynchrony (δ is the maximum message delay after GST). Once GST is reached, all t processes wake up and broadcast their value, for a total of $O(tnL) = O(n^2L)$ exchanged bits; this can happen before they have a chance to receive even a single message! This attack can be considered a "curse" of GST: the *adversarial shift* of correct processes in time creates a (seemingly unavoidable) situation where $\Omega(n^2)$ messages are sent at GST (which in this case include L bits each, for a total of $\Omega(n^2L)$). Figure 2 illustrates the attack.

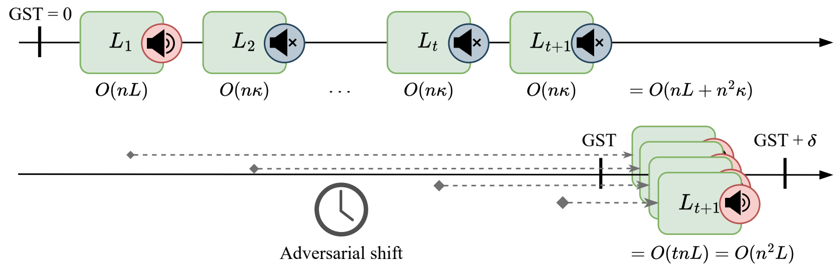

Figure 2 The *adversarial shift* attack on $t+1$ leaders. The first line shows how leaders are optimistically ordered in time by the protocol to avoid redundant broadcasts (the blue speaker circle represents an *avoided* redundant broadcast). The second line shows how leaders can slow down before GST and overlap at GST, making redundant broadcasts (seem) unavoidable.

DARE: Disperse, Agree, REtrieve. In a nutshell, DARE follows three phases:

1. **Dispersal**: Processes attempt to disperse their values and obtain a *proof of dispersal* for any value. This proof guarantees that the value is both (1) valid, and (2) retrievable.
2. **Agreement**: Processes propose a *hash* of the value accompanied by its proof of dispersal to a Byzantine consensus algorithm for small L (e.g., $O(\kappa)$).
3. **Retrieval**: Using the decided hash, processes retrieve the corresponding value. The proof of dispersal ensures Retrieval will succeed and output a valid value.

This architecture is inspired by randomized asynchronous Byzantine algorithms [6, 69] which work with *expected* bit complexity (the worst-case is unbounded in asynchrony [54]). As these algorithms work in expectation, they can rely on randomness to retrieve a value ($\neq \bot$) that is valid after an expected constant number of tries. However, in order to achieve the same effect (i.e., a constant number of retrievals) in the worst case in partial synchrony, DARE must guarantee that the Retrieval protocol *always* outputs a valid value ($\neq \bot$) *a priori*, which shifts the difficulty of the problem almost entirely to the Dispersal phase.

Dispersal. To obtain a proof of dispersal, a natural solution is for the leader to broadcast the value v. Correct processes check the validity of v (i.e., if $\mathsf{valid}(v) = \mathit{true}$), store v for the Retrieval protocol, and produce a partial signature attesting to these two facts. The leader combines the partial signatures into a $(2t+1, n)$-threshold signature (the proof of dispersal), which is sufficient to prove that DARE's Retrieval protocol [44] will output a valid value after the Agreement phase.

However, if leaders use best-effort broadcast [24] (i.e., simultaneously send the value to all other processes), they are still vulnerable to an *adversarial shift* causing $O(n^2L)$ communication. Instead, we do the following. First, we use a *view synchronizer* [78, 20, 21] to group leaders into *views* in a rotating sequence. A view has $\sqrt{n}$ leaders and a sequence has $\sqrt{n}$ views. Leaders of the current view can concurrently broadcast their values while messages of other views are ignored. Second, instead of broadcasting the value simultaneously to all processes, a leader broadcasts the value to different subgroups of $\sqrt{n}$ processes in intervals of δ time (i.e., broadcast to the first subgroup, wait δ time, broadcast to the second subgroup, ...) until all processes have received the value. Neither idea individually is enough to improve over the $O(n^2L)$ term. However, when they are combined, it becomes possible to balance the communication cost of the synchronizer ($O(n^{2.5}\kappa)$ bits), the maximum cost of an *adversarial shift* attack ($O(n^{1.5}L)$ bits), and the broadcast rate to achieve the improved $O(n^{1.5}L + n^{2.5}\kappa)$ bit complexity with asymptotically optimal $O(\delta n)$ latency as shown in Figure 3.

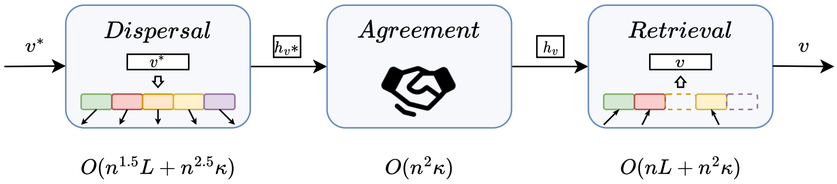

Figure 3 Overview of DARE (Disperse, Agree, REtrieve).

DARE-Stark. As we explained, the main cost of the Dispersal phase is associated with obtaining a dispersal proof that a value is valid. Specifically, it comes from the cost of having to send the entire value (L bits) in a single message.

With Succinct Transparent ARguments of Knowledge (STARKs), we can entirely avoid sending the value in a single message. STARKs allow a process to compute a proof ($O(poly(\kappa))$ bits) of a statement on some value without having to share that value. As an example, a process P_i can send $\langle h, \sigma_{\mathsf{STARK}}\rangle$ to a process P_j, which can use σ_{STARK} to verify the statement "$\exists v : \mathsf{valid}(v) = \mathit{true} \land \mathsf{hash}(v) = h$", all without P_j ever receiving v. As we detail in §6, by carefully crafting a more complex statement, we can modify DARE's Dispersal and Retrieval phases to function with at most $O(poly(\kappa))$ bit-sized messages, obtaining DARE-Stark. This yields the overall near-optimal bit complexity of $O(nL + n^2 poly(\kappa))$. Currently, the main drawback of STARKs is their size and computation time in practice[2], which we hope will improve in the future.

Roadmap. We discuss related work in §2. In §3, we define the system model. We give an overview of DARE in §4. In §5, we detail our Dispersal protocol. We go over DARE-Stark in §6. Lastly, we conclude the paper in §7. Detailed proofs are relegated to the full version of the paper.

2 Related Work

We address the communication complexity of deterministic authenticated Byzantine consensus [65, 25] in partially synchronous distributed systems [52] for large inputs. Here, we discuss existing results in closely related contexts, and provide a brief overview of techniques, tools and building blocks which are often employed to tackle Byzantine consensus.[3]

Asynchrony. In the asynchronous setting, Byzantine agreement is commonly known as Multi-valued Validated Byzantine Agreement, or MVBA [25]. Due to the FLP impossibility result [54], deterministic Byzantine agreement is unsolvable in asynchrony (which implies

[2] The associated constants hidden by the "big O" notation result in computation in the order of seconds, proofs in the hundreds of KB, and memory usage several times greater [53].
[3] We use "consensus" and "agreement" interchangeably.

unbounded worst-case complexity). Hence, asynchronous MVBA solutions focus on expected complexity. This line of work was revitalized by HoneyBadgerBFT [73], the first practical fully asynchronous MVBA implementation. Like most other modern asynchronous MVBA protocols, it leverages randomization via a common coin [76], and it terminates in expected $O(\log n)$ time with an expected bit complexity of $O(n^2L + n^3\kappa \log n)$. [6] improves this to $O(1)$ expected time and $O(n^2L + n^2\kappa)$ expected bits, which is asymptotically optimal with $L, \kappa \in O(1)$. Their result is later extended by [69] to large values, improving the complexity to $O(nL+n^2\kappa)$ expected bits. This matches the best known lower bound [85, 3, 48], assuming $\kappa \in O(1)$.

Extension protocols [79, 56, 57, 58]. An extension protocol optimizes for long inputs via a reduction to the same problem with small inputs (considered an oracle). Using extension protocols, several state-of-the-art results were achieved in the authenticated and unauthenticated models, both in synchronous and fully asynchronous settings for Byzantine consensus, Byzantine broadcast and reliable broadcast [79]. Applying the extension protocol of [79] to [75], synchronous Byzantine agreement can be implemented with optimal resiliency ($t < n/2$) and a bit complexity of $O(nL + n^2\kappa)$. Interestingly, it has been demonstrated that synchronous Byzantine agreement can be implemented with a bit complexity of $O(n(L + \mathit{poly}(\kappa)))$ using randomization [18]. The Dolev-Reischuk bound [48] is not violated in this case since the implementation tolerates a negligible (with κ) probability of failure, whereas the bound holds for deterministic protocols. In asynchrony, by applying the (asynchronous) extension protocol of [79] to [6], the same asymptotic result as [69] is achieved, solving asynchronous MVBA with an expected bit complexity of $O(nL + n^2\kappa)$.

Unconditionally secure Byzantine agreement with large inputs has been addressed by [33, 34] under synchrony and [67] under asynchrony, assuming a common coin (implementable via unconditionally-secure Asynchronous Verifiable Secret Sharing [35]). Despite [61] utilizing erasure codes to alleviate leader bottleneck, and the theoretical construction of [38] with exponential latency, there is, to the best of our knowledge, no viable extension protocol for Byzantine agreement in partial synchrony achieving results similar to ours ($o(n^2L)$).

Error correction. Coding techniques, such as erasure codes [19, 60, 10] or error-correction codes [82, 14], appear in state-of-the-art implementations of various distributed tasks: Asynchronous Verifiable Secret Sharing (AVSS) against a computationally bounded [44, 90, 84] or unbounded [35] adversary, Random Beacon [43], Atomic Broadcast in both the asynchronous [55, 62] and partially synchronous [28] settings, Information-Theoretic (IT) Asynchronous State Machine Replication (SMR) [51], Gradecast in synchrony and Reliable Broadcast in asynchrony [2], Asynchronous Distributed Key Generation (ADKG) [44, 45], Asynchronous Verifiable Information Dispersal (AVID) [9], Byzantine Storage [47, 12, 59], and MVBA [69, 79]. Coding techniques are often used to reduce the worst-case complexity by allowing a group of processes to balance and share the cost of sending a value to an individual (potentially faulty) node and are also used in combination with other techniques, such as commitment schemes [31, 69].

We now list several problems related to or used in solving Byzantine agreement.

Asynchronous Common Subset (ACS). The goal in ACS [14, 15, 50] (also known as Vector Consensus [80, 49, 40]) is to agree on a subset of $n - t$ proposals. When considering a generalization of the validity property, this problem represents the strongest variant of consensus [38]. Atomic Broadcast can be trivially reduced to ACS [49, 25, 73]. There are well-known simple asynchronous constructions that allow for the reduction of ACS to either (1) Reliable Broadcast and Binary Byzantine Agreement [15], or (2) MVBA [25] in the authenticated setting, where the validation predicate requires the output to be a vector of signed inputs from at least $n - t$ parties. The first reduction enables the implementation of ACS with a cubic bit complexity, using the broadcast of [2]. The second reduction could be improved further with a more efficient underlying MVBA protocol, such as DARE-Stark.

Asynchronous Verifiable Information Dispersal (AVID). AVID [26] is a form of "retrievable" broadcast that allows the dissemination of a value while providing a cryptographic proof that it can be retrieved. This primitive can be implemented with a total dispersal cost of $O(L + n^2\kappa)$ bits exchanged and a retrieval cost of $O(L + n\kappa)$ per node, relying only on the existence of collision-resistant hash functions [9]. AVID is similar to our Dispersal and Retrieval phases, but has two key differences. First, AVID's retrieval protocol only guarantees that a valid value will be retrieved if the original process dispersing the information was correct. Second, it is a broadcast protocol, having stricter delivery guarantees for each process. Concretely, if a correct process initiates the AVID protocol, it should eventually disperse its *own* value. In contrast, we only require that a correct process obtains a proof of dispersal for *some* value.

Provable Broadcast (PB) and Asynchronous Provable Dispersal Broadcast (APDB). PB [7] is a primitive used to acquire a succinct proof of external validity. It is similar to our Dispersal phase, including the algorithm itself, but without the provision of a proof of dispersal (i.e., retrievability, only offering proof of validity). The total bit complexity for n PB-broadcasts from distinct processes amounts to $O(n^2L)$. APDB [69] represents an advancement of AVID, drawing inspiration from PB. It sacrifices PB's validity guarantees to incorporate AVID's dissemination and retrieval properties. By leveraging the need to retrieve and validate a value a constant number of times in expectation, [69] attains optimal $O(nL + n^2\kappa)$ expected complexity in asynchrony. However, this approach falls short in the worst-case scenario of a partially synchronous solution, where n reconstructions would cost $\Omega(n^2L)$.

Asynchronous Data Dissemination (ADD). In ADD [44], a subset of $t+1$ correct processes initially share a common L-sized value v, and the goal is to disseminate v to all correct processes, despite the presence of up to t Byzantine processes. The approach of [44] is information-theoretically secure, tolerates up to one-third malicious nodes and has a bit complexity of $O(nL + n^2 \log n)$. (In DARE, we rely on ADD in a "closed-box" manner; see §4.)

3 Preliminaries

Processes. We consider a static set $\mathsf{Process} = \{P_1, P_2, ..., P_n\}$ of $n = 3t + 1$ processes, out of which (at most) $t > 0$ can be Byzantine and deviate arbitrarily from their prescribed protocol. A Byzantine process is said to be *faulty*; a non-faulty process is said to be *correct*. Processes communicate by exchanging messages over an authenticated point-to-point network. Furthermore, the communication network is reliable: if a correct process sends a message to a correct process, the message is eventually received. Processes have local hardware clocks. Lastly, we assume that local steps of processes take zero time, as the time needed for local computation is negligible compared to the message delays.

Partial synchrony. We consider the standard partially synchronous model [52]. For every execution, there exists an unknown Global Stabilization Time (GST) and a positive duration δ such that the message delays are bounded by δ after GST. We assume that δ is known by processes. All correct processes start executing their prescribed protocol by GST. The hardware clocks of processes may drift arbitrarily before GST, but do not drift thereafter. We underline that our algorithms require minimal changes to preserve their correctness even if δ is unknown (these modifications are specified in Appendix C.2), although their complexity might be higher.

Cryptographic primitives. Throughout the paper, $\mathsf{hash}(\cdot)$ denotes a collision-resistant hash function. The codomain of the aforementioned $\mathsf{hash}(\cdot)$ function is denoted by $\mathsf{Hash\_Value}$.

Moreover, we assume a (k, n)-threshold signature scheme [83], where $k = n - t = 2t + 1$. In this scheme, each process holds a distinct private key, and there is a single public key. Each process P_i can use its private key to produce a partial signature for a message m by invoking $\mathsf{share\_sign}_i(m)$. A set of partial signatures S for a message m from k distinct processes can be combined into a single threshold signature for m by invoking $\mathsf{combine}(S)$; a threshold signature for m proves that k processes have (partially) signed m. Furthermore, partial and threshold signatures can be verified: given a message m and a signature Σ_m, $\mathsf{verify\_sig}(m, \Sigma_m)$ returns *true* if and only if Σ_m is a valid signature for m. Where appropriate, the verifications are left implicit. We denote by $\mathsf{P\_Signature}$ and $\mathsf{T\_Signature}$ the set of partial and threshold signatures, respectively. The size of cryptographic objects (i.e., hashes, signatures) is denoted by κ; we assume that $\kappa > \log n$.[4]

Reed-Solomon codes [82]. Our algorithms rely on Reed-Solomon (RS) codes [81]. Concretely, DARE utilizes (in a "closed-box" manner) an algorithm which internally builds upon error-correcting RS codes. DARE-Stark directly uses RS erasure codes (no error correction is required).

We use $\mathsf{encode}(\cdot)$ and $\mathsf{decode}(\cdot)$ to denote RS' encoding and decoding algorithms. In a nutshell, $\mathsf{encode}(\cdot)$ takes a value v, chunks it into the coefficients of a polynomial of degree t (the maximum number of faults), and outputs n (the total number of processes) evaluations of the polynomial (RS symbols); Symbol denotes the set of RS symbols. $\mathsf{decode}(\cdot)$ takes a set of $t + 1$ RS symbols S and interpolates them into a polynomial of degree t, whose coefficients are concatenated and output.

Complexity of Byzantine consensus. Let $\mathsf{Consensus}$ be a partially synchronous Byzantine consensus algorithm, and let $\mathcal{E}(\mathsf{Consensus})$ denote the set of all possible executions. Let $\alpha \in \mathcal{E}(\mathsf{Consensus})$ be an execution, and $t_d(\alpha)$ be the first time by which all correct processes have decided in α. The bit complexity of α is the total number of bits sent by correct processes during the time period $[\text{GST}, \infty)$. The latency of α is $\max(0, t_d(\alpha) - \text{GST})$.

The *bit complexity* of $\mathsf{Consensus}$ is defined as

$$\max_{\alpha \in \mathcal{E}(\mathsf{Consensus})} \left\{ \text{bit complexity of } \alpha \right\}.$$

[4] For $\kappa \leq \log n$, $t \in O(n)$ faulty processes would have computational power exponential in κ, breaking cryptographic hardness assumptions.

Similarly, the *latency* of Consensus is defined as

$$\max_{\alpha \in \mathcal{E}(\mathsf{Consensus})} \Big\{ \text{latency of } \alpha \Big\}.$$

4 DARE

This section presents DARE (Disperse, Agree, REtrieve), which is composed of three algorithms:

1. Disperser, which disperses the proposals;
2. Agreement, which ensures agreement on the hash of a previously dispersed proposal; and
3. Retriever, which rebuilds the proposal corresponding to the agreed-upon hash.

We start by introducing the aforementioned building blocks (§4.1). Then, we show how they are composed into DARE (§4.2). Finally, we prove the correctness and complexity of DARE (§4.3).

4.1 Building Blocks: Overview

In this subsection, we formally define the three building blocks of DARE. Concretely, we define their interface and properties, as well as their complexity.

4.1.1 Disperser

Interface & properties. Disperser solves a problem similar to that of AVID [26]. In a nutshell, each correct process aims to disperse its value to all correct processes: eventually, all correct processes acquire a proof that a value with a certain hash has been successfully dispersed.

Concretely, Disperser exposes the following interface:

- **request** $\mathsf{disperse}(v \in \mathsf{Value})$: a process disperses a value v; each correct process invokes $\mathsf{disperse}(v)$ exactly once and only if $\mathsf{valid}(v) = \mathit{true}$.
- **indication** $\mathsf{acquire}(h \in \mathsf{Hash\_Value}, \Sigma_h \in \mathsf{T\_Signature})$: a process acquires a pair (h, Σ_h).

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 Retriever; see §4.1.3 and Algorithm 1.) Disperser ensures the following:

- *Integrity:* If a correct process acquires a hash-signature pair (h, Σ_h), then $\mathsf{verify\_sig}(h, \Sigma_h) = \mathit{true}$.
- *Termination:* Every correct process eventually acquires at least one hash-signature pair.
- *Redundancy:* Let a correct process obtain a threshold signature Σ_h such that $\mathsf{verify\_sig}(h, \Sigma_h) = \mathit{true}$, for some hash value h. Then, (at least) $t + 1$ correct processes have obtained a value v such that (1) $\mathsf{hash}(v) = h$, and (2) $\mathsf{valid}(v) = \mathit{true}$.

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

Complexity. Disperser exchanges $O(n^{1.5}L + n^{2.5}\kappa)$ bits after GST. Moreover, it terminates in $O(n)$ time after GST.

Implementation. The details on Disperser's implementation are relegated to §5.

4.1.2 Agreement

Interface & properties. Agreement is a Byzantine consensus algorithm.[5] In Agreement, processes propose and decide pairs ($h \in \mathsf{Hash\_Value}, \Sigma_h \in \mathsf{T\_Signature}$); moreover, $\mathsf{valid}(h, \Sigma_h) \equiv \mathsf{verify\_sig}(h, \Sigma_h)$.

Complexity. Agreement achieves $O(n^2\kappa)$ bit complexity and $O(n)$ latency.

Implementation. We "borrow" the implementation from [36]. In brief, Agreement is a "leader-based" consensus algorithm whose computation unfolds in views. Each view has a single leader, and it employs a "leader-to-all, all-to-leader" communication pattern. Agreement's safety relies on standard techniques [89, 30, 23, 66]: (1) quorum intersection (safety within a view), and (2) "locking" mechanism (safety across multiple views). As for liveness, Agreement guarantees termination once all correct processes are in the same view (for "long enough" time) with a correct leader. (For full details on Agreement, see [36].)

4.1.3 Retriever

Interface & properties. In Retriever, each correct process starts with either (1) some value, or (2) $\perp$. Eventually, all correct processes output the same value. Formally, Retriever exposes the following interface:

- **request** $\mathsf{input}(v \in \mathsf{Value} \cup \{\perp\})$: a process inputs a value or $\perp$; each correct process invokes $\mathsf{input}(\cdot)$ exactly once. Moreover, the following is assumed:
  - No two correct processes invoke $\mathsf{input}(v_1 \in \mathsf{Value})$ and $\mathsf{input}(v_2 \in \mathsf{Value})$ with $v_1 \neq v_2$.
  - At least $t+1$ correct processes invoke $\mathsf{input}(v \in \mathsf{Value})$ (i.e., $v \neq \perp$).
- **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. Retriever exchanges $O(nL + n^2 \log n)$ bits after GST (and before every correct process outputs a value). Moreover, Retriever terminates in $O(1)$ time after GST.

Implementation. Retriever's implementation is "borrowed" from [44]. In summary, Retriever relies on Reed-Solomon codes [82] to encode the input value $v \neq \perp$ into n symbols. Each correct process Q which inputs $v \neq \perp$ to Retriever encodes v into n RS symbols $s_1, s_2, ..., s_n$. Q sends each RS symbol s_i to the process P_i. When P_i receives $t+1$ identical RS symbols s_i, P_i is sure that s_i is a "correct" symbol (i.e., it can be used to rebuild v) as it was computed by at least one correct process. At this moment, P_i broadcasts s_i. Once each correct process P receives $2t+1$ (or more) RS symbols, P tries to rebuild v (with some error-correction). (For full details on Retriever, see [44].)

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

Algorithm 1 DARE: Pseudocode (for process P_i).

1: **Uses:**
2: ▷ bits: $O(n^{1.5}L + n^{2.5}\kappa)$, latency: $O(n)$ (see §5)
3: DISPERSER, **instance** *disperser*
4: ▷ bits: $O(n^2\kappa)$, latency: $O(n)$ (see [36])
5: AGREEMENT, **instance** *agreement*
6: ▷ bits: $O(nL + n^2 \log n)$, latency: $O(1)$ (see [44])
7: RETRIEVER, **instance** *retriever*
8: **upon** propose($v_i \in$ Value):
9: **invoke** *disperser*.disperse(v_i)
10: **upon** *disperser*.acquire($h_i \in$ Hash_Value, $\Sigma_i \in$ T_Signature):
11: **invoke** *agreement*.propose(h_i, Σ_i)
12: **upon** *agreement*.decide($h \in$ Hash_Value, $\Sigma_h \in$ T_Signature): ▷ P_i obtains Σ_h
13: $v \leftarrow$ an obtained value such that hash(v) $= h$ (if such a value was not obtained, $v = \perp$)
14: **invoke** *retriever*.input(v)
15: **upon** *retriever*.output(Value v'):
16: **trigger** decide(v')

4.2 Pseudocode

Algorithm 1 gives DARE's pseudocode. We explain it from the perspective of a correct process P_i. An execution of DARE consists of three phases (each of which corresponds to one building block):

1. *Dispersal:* Process P_i disperses its proposal v_i using DISPERSER (line 9). Eventually, P_i acquires a hash-signature pair (h_i, Σ_i) (line 10) due to the termination property of DISPERSER.
2. *Agreement:* Process P_i proposes the previously acquired hash-signature pair (h_i, Σ_i) to AGREEMENT (line 11). As AGREEMENT satisfies termination and agreement, all correct processes eventually agree on a hash-signature pair (h, Σ_h) (line 12).
3. *Retrieval:* Once process P_i decides (h, Σ_h) from AGREEMENT, it checks whether it has previously obtained a value v with hash$(v) = h$ (line 13). If it has, P_i inputs v to RETRIEVER; otherwise, P_i inputs $\perp$ (line 14). The required preconditions for RETRIEVER are met:
   - No two correct processes input different non-$\perp$ values to RETRIEVER as hash$(\cdot)$ is collision-resistant.
   - At least $(t + 1)$ correct processes input a value (and not $\perp$) to RETRIEVER. Indeed, as Σ_h is obtained by a correct process, $t + 1$ correct processes have obtained a value $v \neq \perp$ with hash$(v) = h$ (due to redundancy of DISPERSER), and all of these processes input v.

   Therefore, all correct processes (including P_i) eventually output the same value v' from RETRIEVER (due to the termination property of RETRIEVER; line 15), which represents the decision of DARE (line 16). Note that $v' = v \neq \perp$ due to the validity of RETRIEVER.

4.3 Proof of Correctness & Complexity

We start by proving the correctness of DARE.

▶ **Theorem 1.** *DARE is correct.*

Proof. Every correct process starts the dispersal of its proposal (line 9). Due to the termination property of DISPERSER, every correct process eventually acquires a hash-signature pair (line 10). Hence, every correct process eventually proposes to AGREEMENT (line 11), which implies that every correct process eventually decides the same hash-signature pair (h, Σ_h) from AGREEMENT (line 12) due to the agreement and termination properties of AGREEMENT.

As (h, Σ_h) is decided by all correct processes, at least $t+1$ correct processes P_i have obtained a value v such that (1) $\mathsf{hash}(v) = h$, and (2) $\mathsf{valid}(v) = \mathit{true}$ (due to the redundancy property of DISPERSER). Therefore, all of these correct processes input v to RETRIEVER (line 14). Moreover, no correct process inputs a different value (as $\mathsf{hash}(\cdot)$ is collision-resistant). Thus, the conditions required by RETRIEVER are met, which implies that all correct processes eventually output the same valid value (namely, v) from RETRIEVER (line 15), and decide it (line 16). ◀

Next, we prove the complexity of DARE.

▶ **Theorem 2.** *DARE achieves $O(n^{1.5}L + n^{2.5}\kappa)$ bit complexity and $O(n)$ latency.*

Proof. As DARE is a sequential composition of its building blocks, its complexity is the sum of the complexities of (1) DISPERSER, (2) AGREEMENT, and (3) RETRIEVER. Hence, the bit complexity is

$$\underbrace{O(n^{1.5}L + n^{2.5}\kappa)}_{\text{DISPERSER}} + \underbrace{O(n^2\kappa)}_{\text{AGREEMENT}} + \underbrace{O(nL + n^2\log n)}_{\text{RETRIEVER}} = O(n^{1.5}L + n^{2.5}\kappa).$$

Similarly, the latency is $O(n)$. ◀

5 Disperser: Implementation & Analysis

This section focuses on DISPERSER. Namely, we present its implementation (§5.1), and (informally) analyze its correctness and worst-case complexity (§5.2). Formal proofs can be found in the full version of the paper [37]. An analysis of the good (common) case can be found in Appendix C.1.

5.1 Implementation

DISPERSER's pseudocode is given in Algorithm 2. In essence, each execution unfolds in *views*, where each view has X *leaders* ($0 < X \leq n$ is a generic parameter); the set of all views is denoted by View. Given a view V, $\mathsf{leaders}(V)$ denotes the X-sized set of leaders of the view V. In each view, a leader disperses its value to Y-sized groups of processes ($0 < Y \leq n$ is a generic parameter) at a time (line 14), with a δ-waiting step in between (line 15). Before we thoroughly explain the pseudocode, we introduce SYNC, DISPERSER's view synchronization [36, 89, 66] algorithm.

Sync. Its responsibility is to bring all correct processes to the same view with a correct leader for (at least) $\Delta = \delta\frac{n}{Y} + 3\delta$ time. Precisely, SYNC exposes the following interface:

- **indication** $\mathsf{advance}(V \in \mathsf{View})$: a process enters a new view V.

SYNC guarantees *eventual synchronization*: there exists a time $\tau_{sync} \geq$ GST (*synchronization time*) such that (1) all correct processes are in the same view V_{sync} (*synchronization view*) from time τ_{sync} to (at least) time $\tau_{sync} + \Delta$, and (2) V_{sync} has a correct leader. We denote by V^*_{sync} the smallest synchronization view, whereas τ^*_{sync} denotes the first synchronization time. Similarly, V_{max} denotes the greatest view entered by a correct process before GST.[6]

[6] When such a view does not exist, $V_{max} = 0$

The implementation of Sync (see Appendix A) is highly inspired by RareSync, a view synchronization algorithm introduced in [36]. In essence, when a process enters a new view, it stays in the view for $O(\Delta) = O(\frac{n}{Y})$ time. Once it wishes to proceed to the next view, the process engages in an "all-to-all" communication step (which exchanges $O(n^2\kappa)$ bits); this step signals the end of the current view, and the beginning of the next one. Throughout views, leaders are rotated in a round-robin manner: each process is a leader for exactly one view in any sequence of $\frac{n}{X}$ consecutive views. As $O(\frac{n}{X})$ views (after GST) are required to reach a correct leader, Sync exchanges $O(\frac{n}{X}) \cdot O(n^2\kappa) = O(\frac{n^3\kappa}{X})$ bits (before synchronization, i.e., before $\tau^*_{sync} + \Delta$); since each view takes $O(\frac{n}{Y})$ time, synchronization is ensured within $O(\frac{n}{X}) \cdot O(\frac{n}{Y}) = O(\frac{n^2}{XY})$ time.

Disperser relies on the following properties of Sync (along with eventual synchronization):

- *Monotonicity:* Any correct process enters monotonically increasing views.
- *Stabilization:* Any correct process enters a view $V \geq V_{max}$ by time $\text{GST} + 3\delta$.
- *Limited entrance:* In the time period $[\text{GST}, \text{GST} + 3\delta)$, any correct process enters $O(1)$ views.
- *Overlapping:* For any view $V > V_{max}$, all correct processes overlap in V for (at least) Δ time.
- *Limited synchronization view:* $V^*_{sync} - V_{max} = O(\frac{n}{X})$.
- *Complexity:* Sync exchanges $O(\frac{n^3\kappa}{X})$ bits during the time period $[\text{GST}, \tau^*_{sync} + \Delta]$, and it synchronizes all correct processes within $O(\frac{n^2}{XY})$ time after GST ($\tau^*_{sync} + \Delta - \text{GST} = O(\frac{n^2}{XY})$).

The aforementioned properties of Sync are formally proven in Appendix A.

Algorithm description. Correct processes transit through views based on Sync's indications (line 10): when a correct process receives $\mathsf{advance}(V)$ from Sync, it stops participating in the previous view and starts participating in V.

Once a correct leader P_l enters a view V, it disperses its proposal via dispersal messages. As already mentioned, P_l sends its proposal to Y-sized groups of processes (line 14) with a δ-waiting step in between (line 15). When a correct (non-leader) process P_i (which participates in the view V) receives a dispersal message from P_l, P_i checks whether the dispersed value is valid (line 17). If it is, P_i partially signs the hash of the value, and sends it back to P_l (line 20). When P_l collects $2t+1$ ack messages, it (1) creates a threshold signature for the hash of its proposal (line 24), and (2) broadcasts the signature (along with the hash of its proposal) to all processes via a confirm message (line 25). Finally, when P_l (or any other correct process) receives a confirm message (line 27), it (1) acquires the received hash-signature pair (line 28), (2) disseminates the pair to "help" the other processes (line 29), and (3) stops executing Disperser (line 30).

5.2 Analysis

Correctness. Once all correct processes synchronize in the view V^*_{sync} (the smallest synchronization view), all correct processes acquire a hash-signature pair. Indeed, $\Delta = \delta\frac{n}{Y} + 3\delta$ time is sufficient for a correct leader $P_l \in \mathsf{leaders}(V^*_{sync})$ to (1) disperse its proposal $proposal_l$ to all processes (line 14), (2) collect $2t+1$ partial signatures for $h = \mathsf{hash}(proposal_l)$ (line 23), and (3) disseminate a threshold signature for h (line 25). When a correct process receives the aforementioned threshold signature (line 27), it acquires the hash-signature pair (line 28) and stops executing Disperser (line 30).

Algorithm 2 DISPERSER: Pseudocode (for process P_i).

1: **Uses:**
2: SYNC, **instance** *sync* ▷ ensures a $\Delta = \delta\frac{n}{Y} + 3\delta$ overlap in a view with a correct leader
3: **upon** init:
4: Value $proposal_i \leftarrow \bot$
5: Integer $received\_acks_i \leftarrow 0$
6: Map(Hash_Value $\rightarrow$ Value) $obtained\_values_i \leftarrow$ empty
7: **upon** disperse(Value v_i):
8: $proposal_i \leftarrow v_i$
9: **start** *sync*
10: **upon** *sync*.advance(View V): ▷ P_i stops participating in the previous view
11: ▷ First part of the view
12: **if** $P_i \in$ leaders(V):
13: **for** Integer $k \leftarrow 1$ to $\frac{n}{Y}$:
14: **send** $\langle$DISPERSAL, $proposal_i\rangle$ to $P_{(k-1)Y+1}, P_{(k-1)Y+2}, ..., P_{kY}$
15: **wait** δ time
16: every process:
17: **upon** reception of $\langle$DISPERSAL, Value $v_j\rangle$ from process $P_j \in$ leaders(V) and valid(v_j) $= true$:
18: Hash_Value $h \leftarrow$ hash(v_j)
19: $obtained\_values_i[h] \leftarrow v_j$
20: **send** $\langle$ACK, share_sign$_i(h)\rangle$ to P_j
21: ▷ Second part of the view
22: **if** $P_i \in$ leaders(V):
23: **upon** exists Hash_Value h such that $\langle$ACK, $h, \cdot\rangle$ has been received from $2t+1$ processes:
24: T_Signature $\Sigma_h \leftarrow$ combine$\big(\{$P_Signature $sig \mid sig$ is received in the ACK messages$\}\big)$
25: **broadcast** $\langle$CONFIRM, $h, \Sigma_h\rangle$
26: every process:
27: **upon** reception of $\langle$CONFIRM, Hash_Value h, T_Signature $\Sigma_h\rangle$ and verify_sig(h, Σ_h) $= true$:
28: **trigger** acquire(h, Σ_h)
29: **broadcast** $\langle$CONFIRM, $h, \Sigma_h\rangle$
30: **stop** executing DISPERSER (and SYNC)

Complexity. DISPERSER terminates once all correct processes are synchronized in a view with a correct leader. The synchronization is ensured in $O(\frac{n^2}{XY})$ time after GST (as $\tau^*_{sync} + \Delta - \text{GST} = O(\frac{n^2}{XY})$). Hence, DISPERSER terminates in $O(\frac{n^2}{XY})$ time after GST.

Let us analyze the number of bits DISPERSER exchanges. Any execution of DISPERSER can be separated into two post-GST periods: (1) *unsynchronized*, from GST until GST $+ 3\delta$, and (2) *synchronized*, from GST $+ 3\delta$ until $\tau^*_{sync} + \Delta$. First, we study the number of bits correct processes send via DISPERSAL, ACK and CONFIRM message in the aforementioned periods:

- Unsynchronized period: Due to the δ-waiting step (line 15), each correct process sends DISPERSAL messages (line 14) to (at most) $3 = O(1)$ Y-sized groups. Hence, each correct process sends $O(1) \cdot O(Y) \cdot L = O(YL)$ bits through DISPERSAL messages.

  Due to the limited entrance property of SYNC, each correct process enters $O(1)$ views during the unsynchronized period. In each view, each correct process sends (at most) $O(X)$ ACK messages (one to each leader; line 20) and $O(n)$ CONFIRM messages (line 25). As each ACK and CONFIRM message carries κ bits, all correct processes send

$$n \cdot \big(\underbrace{O(YL)}_{\text{DISPERSAL}} + \underbrace{O(X\kappa)}_{\text{ACK}} + \underbrace{O(n\kappa)}_{\text{CONFIRM}}\big)$$
$$= O(nYL + n^2\kappa) \text{ bits via DISPERSAL, ACK and CONFIRM messages.}$$

- Synchronized period: Recall that all correct processes acquire a hash-signature pair (and stop executing DISPERSER) by time $\tau^*_{sync} + \Delta$, and they do so in the view V^*_{sync}. As correct processes enter monotonically increasing views, no correct process enters a view greater than V^*_{sync}.
  By the stabilization property of SYNC, each correct process enters a view $V \geq V_{max}$ by time $\text{GST} + 3\delta$. Moreover, until $\tau^*_{sync} + \Delta$, each correct process enters (at most) $O(\frac{n}{X})$ views (due to the limited synchronization view and monotonicity properties of SYNC). Importantly, no correct leader exists in any view V with $V_{max} < V < V^*_{sync}$; otherwise, $V = V^*_{sync}$ as processes overlap for Δ time in V (due to the overlapping property of SYNC). Hence, for each view V with $V_{max} < V < V^*_{sync}$, all correct processes send $O(nX\kappa)$ bits (all through ACK messages; line 20). In V_{max} and V^*_{sync}, all correct processes send (1) $2 \cdot O(XnL)$ bits through DISPERSAL messages (line 14), (2) $2 \cdot O(nX\kappa)$ bits through ACK messages (line 20), and (3) $2 \cdot O(Xn\kappa)$ bits through CONFIRM messages (line 25). Therefore, all correct processes send
  $$\underbrace{O(\frac{n}{X})}_{V^*_{sync} - V_{max}} \cdot \underbrace{O(nX\kappa)}_{\text{ACK}} + \underbrace{O(XnL)}_{\text{DISPERSAL in } V_{max} \text{ and } V^*_{sync}} + \underbrace{O(nX\kappa)}_{\text{ACK in } V_{max} \text{ and } V^*_{sync}} + \underbrace{O(Xn\kappa)}_{\text{CONFIRM in } V_{max} \text{ and } V^*_{sync}}$$
  $= O(nXL + n^2\kappa)$ bits via DISPERSAL, ACK and CONFIRM messages.

We cannot neglect the complexity of SYNC, which exchanges $O(\frac{n^3\kappa}{X})$ bits during the time period $[\text{GST}, \tau^*_{sync} + \Delta]$. Hence, the total number of bits DISPERSER exchanges is

$$\underbrace{O(nYL + n^2\kappa)}_{\text{unsynchronized period}} + \underbrace{O(nXL + n^2\kappa)}_{\text{synchronized period}} + \underbrace{O(\frac{n^3\kappa}{X})}_{\text{SYNC}} = O(nYL + nXL + \frac{n^3\kappa}{X}).$$

With $X = Y = \sqrt{n}$, DISPERSER terminates in optimal $O(n)$ time, and exchanges $O(n^{1.5}L + n^{2.5}\kappa)$ bits. Our analysis is illustrated in Figure 4.

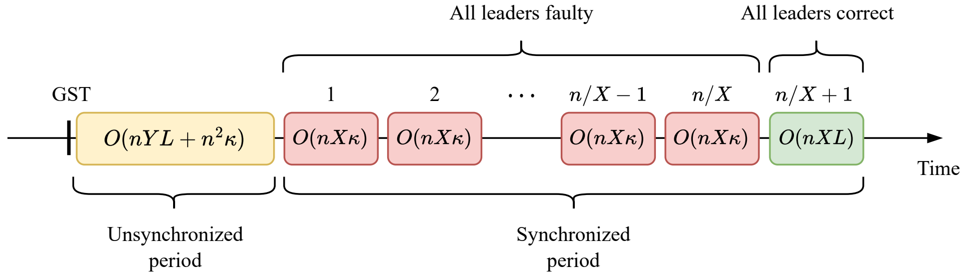

Figure 4 Illustration of DISPERSER's bit complexity.

6 DARE-Stark

In this section, we present DARE-STARK, a variant of DARE which relies on STARK proofs. Importantly, DARE-STARK achieves $O(nL + n^2 poly(\kappa))$ bit complexity, nearly tight to the $\Omega(nL + n^2)$ lower bound, while preserving optimal $O(n)$ latency.

First, we revisit DISPERSER, pinpointing its complexity on proving RS encoding (§6.1). We then present DARE-STARK, which uses STARKs for provable RS encoding, thus improving on DARE's complexity (§6.2). For a brief overview on STARKs, a cryptographic primitive providing succinct proofs of knowledge, we invite the interested reader to Appendix B.

6.1 Revisiting DARE: What Causes Disperser's Complexity?

Recall that DISPERSER exchanges $O(n^{1.5}L + n^{2.5}\kappa)$ bits. This is due to a fundamental requirement of RETRIEVER: at least $t+1$ correct processes must have obtained the value v by the time AGREEMENT decides $h = \mathsf{hash}(v)$. RETRIEVER leverages this requirement to prove the correct encoding of RS symbols. In brief (as explained in §4.1.3): (1) every correct process P that obtained $v \neq \bot$ encodes it in n RS symbols $s_1, \dots, s_n$; (2) P sends each s_i to P_i; (3) upon receiving $t+1$ identical copies of s_i, P_i can trust s_i to be the i-th RS symbol for v (note that s_i can be trusted only because it was produced by at least one correct process – nothing else proves s_i's relationship to v!); (4) every correct process P_i disseminates s_i, enabling the reconstruction of v by means of error-correcting decoding. In summary, DARE bottlenecks on DISPERSER, and DISPERSER's complexity is owed to the need to prove the correct encoding of RS symbols in RETRIEVER. Succinct arguments of knowledge (such as STARKs), however, allow to publicly prove the relationship between an RS symbol and the value it encodes, eliminating the need to disperse the entire value to $t+1$ correct processes – a dispersal of provably correct RS symbols suffices. DARE-STARK builds upon this idea.

6.2 Implementation

Provably correct encoding. At its core, DARE-STARK uses STARKs to attest the correct RS encoding of values. For every $i \in [1, n]$, we define $\mathsf{shard}_i(\cdot)$ by

$$\mathsf{shard}_i(v \in \mathsf{Value}) = \begin{cases} \big(\mathsf{hash}(v), \mathsf{encode}_i(v)\big), & \text{if and only if } \mathsf{valid}(v) = \textit{true} \\ \bot, & \text{otherwise,} \end{cases} \tag{1}$$

where $\mathsf{encode}_i(v)$ represents the i-th RS symbol obtained from $\mathsf{encode}(v)$ (see §3). We use $\mathsf{proof}_i(v)$ to denote the STARK proving the correct computation of $\mathsf{shard}_i(v)$. The design and security of DARE-STARK rests on the following theorem.

▶ **Theorem 3.** *Let $i_1, \dots, i_{t+1}$ be distinct indices in $[1, n]$. Let h be a hash, let $s_1, \dots, s_{t+1}$ be RS symbols, let $stark_1, \dots, stark_{t+1}$ be STARK proofs such that, for every $k \in [1, t+1]$, $stark_k$ proves knowledge of some (undisclosed) v_k such that $\mathsf{shard}_{i_k}(v_k) = (h, s_k)$. We have that*

$$v = \mathsf{decode}(\{s_1, \dots, s_k\})$$

satisfies $\mathsf{valid}(v) = true$ and $\mathsf{hash}(v) = h$.

Proof. For all k, by the correctness of $stark_k$ and Equation (1), we have that (1) $h = \mathsf{hash}(v_k)$, (2) $s_k = \mathsf{encode}_{i_k}(v_k)$, and (3) $\mathsf{valid}(v_k) = true$. By the collision-resistance of $\mathsf{hash}(\cdot)$, for all k, k', we have $v_k = v'_k$. By the definition of $\mathsf{encode}(\cdot)$ and $\mathsf{decode}(\cdot)$, we then have

$$v = \mathsf{decode}(\{s_1, \dots, s_k\}) = v_1 = \ldots = v_{t+1},$$

which implies that $\mathsf{valid}(v) = true$ and $\mathsf{hash}(v) = h$. ◀

Algorithm description. The pseudocode of DARE-STARK is presented in Algorithm 3 from the perspective of a correct process P_i. Similarly to DARE, DARE-STARK unfolds in three phases:

1. *Dispersal:* Upon proposing a value v_i (line 9), P_i sends (line 14) to each process P_k (1) $(h_k, s_k) = \mathsf{shard}_k(v_i)$ (computed at line 12), and (2) $stark_k = \mathsf{proof}_k(v_i)$ (computed at line 13). In doing so (see Theorem 3), P_i proves to P_k that $h_k = \mathsf{hash}(v_i)$ is the hash of a

Algorithm 3 DARE-Stark: Pseudocode (for process P_i).

1: **Uses:**
2: Agreement, **instance** *agreement*
3: **upon** init:
4: Hash_Value $proposed\_hash_i \leftarrow \bot$
5: $\mathsf{Map}\big(\mathsf{Hash\_Value} \rightarrow (\mathsf{Symbol}, \mathsf{STARK})\big)$ $proposal\_shards_i \leftarrow$ empty
6: $\mathsf{Map}\big(\mathsf{Hash\_Value} \rightarrow \mathsf{Set}(\mathsf{Symbol})\big)$ $decision\_symbols_i \leftarrow$ empty
7: Bool $decided_i \leftarrow false$
8: ▷ Dispersal
9: **upon** propose(Value v_i):
10: $proposed\_hash_i \leftarrow \mathsf{hash}(v_i)$
11: **for** Integer $k \leftarrow 1$ to n:
12: $(\mathsf{Hash}\ h_k, \mathsf{Symbol}\ s_k) \leftarrow \mathsf{shard}_k(v_i)$
13: STARK $stark_k \leftarrow \mathsf{proof}_k(v_i)$
14: **send** $\langle$DISPERSAL$, h_k, s_k, stark_k\rangle$ to P_k
15: **upon** reception of $\langle$DISPERSAL$, \mathsf{Hash_Value}\ h, \mathsf{Symbol}\ s, \mathsf{STARK}\ stark\rangle$ from process P_j and $stark$ proves $\mathsf{shard}_i(?) = (h, s)$:
16: $proposal\_shards_i[h] \leftarrow (s, stark)$
17: **send** $\langle$ACK$, \mathsf{share_sign}_i(h)\rangle$ to P_j
18: ▷ Agreement
19: **upon** $\langle$ACK$, \mathsf{P_Signature}\ sig\rangle$ is received from $2t+1$ processes:
20: T_Signature $\Sigma \leftarrow \mathsf{combine}\big(\{sig \mid sig$ is received in the ACK messages$\}\big)$
21: **invoke** $agreement.\mathsf{propose}(proposed\_hash_i, \Sigma)$
22: **upon** $agreement.\mathsf{decide}(\mathsf{Hash\_Value}\ h, \mathsf{T\_Signature}\ \Sigma)$ with $proposal\_shards_i[h] \neq \bot$:
23: $(\mathsf{Symbol}\ s, \mathsf{STARK}\ stark) \leftarrow proposal\_shards_i[h]$
24: **broadcast** $\langle$RETRIEVE$, h, s, stark\rangle$
25: ▷ Retrieval
26: **upon** reception of $\langle$RETRIEVE$, \mathsf{Hash_Value}\ h, \mathsf{Symbol}\ s, \mathsf{STARK}\ stark\rangle$ from process P_j and $stark$ proves $\mathsf{shard}_j(?) = (h, s)$:
27: $decision\_symbols_i[h] \leftarrow decision\_symbols_i[h] \cup \{s\}$
28: **upon** (1) exists Hash_Value h such that $decision\_symbols_i[h]$ has $t+1$ elements, and (2) $decided_i = false$:
29: $decided_i \leftarrow true$
30: **trigger** $\mathsf{decide}\big(\mathsf{decode}(decision\_symbols_i[h])\big)$

valid proposal, whose k-th RS symbol is $\mathsf{encode}_k(v_i)$. P_k checks $stark_k$ against (h_k, s_k) (line 15), stores $(s_k, stark_k)$ (line 16), and sends a partial signature for h_k back to P_i (line 17).

2. *Agreement:* Having collected a threshold signature Σ for $\mathsf{hash}(v_i)$ (line 20), P_i proposes $(\mathsf{hash}(v_i), \Sigma)$ to Agreement (line 21).
3. *Retrieval:* Upon deciding a hash h from Agreement (line 22), P_i broadcasts (if available) the i-th RS symbol for h, along with the relevant proof (line 24). Upon receiving $t+1$ symbols S for the same hash (line 28), P_i decides $\mathsf{decode}(S)$ (line 30).

Analysis. Upon proposing a value v_i (line 9), a correct process P_i sends $\mathsf{shard}_k(v_i)$ and $\mathsf{proof}_k(v_i)$ to each process P_k (line 14). Checking $\mathsf{proof}_k(v_i)$ against $\mathsf{shard}_k(v_i)$ (line 15), P_k confirms having received the k-th RS symbol for v_i (note that this does not require the transmission of v_i, just $\mathsf{hash}(v_i)$). As $2t+1$ processes are correct, P_i is guaranteed to eventually gather a $(2t+1)$-threshold signature Σ for $\mathsf{hash}(v_i)$ (line 20). Upon doing so, P_i proposes $(\mathsf{hash}(v_i), \Sigma)$ to Agreement (line 21). Since every correct process eventually proposes a value to Agreement, every correct process eventually decides some hash h from Agreement (line 22). Because $2t+1$ processes signed h, at least $t+1$ correct processes (without loss of generality, $P_1, \ldots, P_{t+1}$) received a correctly encoded RS-symbol for h. More precisely, for every $k \in [1, t+1]$, P_k received and stored the k-th RS symbol encoded from the pre-image v of h. Upon deciding from Agreement, each process P_k broadcasts its RS symbol, along with the relevant proof (line 24). Because at most t processes are faulty, no correct process receives $t+1$ RS symbols pertaining to a hash other than h. As $P_1, \dots, P_{t+1}$ all broadcast their symbols and proofs, eventually every correct process collects $t+1$ (provably correct) RS symbols S pertaining to h (line 28), and decides $\mathsf{decode}(S)$ (line 30). By Theorem 3, every correct process eventually decides the same valid value v (with $h = \mathsf{hash}(v)$).

Concerning bit complexity, throughout an execution of DARE-Stark, a correct process engages once in Agreement (which exchanges $O(n^2\kappa)$ bits in total) and sends: (1) n dispersal messages, each of size $O(\frac{L}{n} + poly(\kappa))$, (2) n ack messages, each of size $O(\kappa)$, and (3) n retrieve messages, each of size $O(\frac{L}{n} + poly(\kappa))$. Therefore, the bit complexity of DARE-Stark is $O(nL + n^2 poly(\kappa))$. As for the latency, it is $O(n)$ (due to the linear latency of Agreement).

7 Concluding Remarks

This paper introduces DARE (Disperse, Agree, REtrieve), the first partially synchronous Byzantine agreement algorithm on values of L bits with better than $O(n^2L)$ bit complexity and sub-exponential latency. DARE achieves $O(n^{1.5}L + n^{2.5}\kappa)$ bit complexity (κ is the security parameter) and optimal $O(n)$ latency, which is an effective $\sqrt{n}$ factor bit-improvement for $L \geq n\kappa$ (typical in practice). DARE achieves its complexity in two steps. First, DARE decomposes problem of agreeing on large values (L bits) into three sub-problems: (1) value dispersal, (2) validated agreement on small values ($O(\kappa)$), and (3) value retrieval. (DARE effectively acts as an extension protocol for Byzantine agreement.) Second, DARE's novel dispersal algorithm solves the main challenge, value dispersal, using only $O(n^{1.5}L)$ bits and linear latency.

Moreover, we prove that the lower bound of $\Omega(nL + n^2)$ is near-tight by matching it near-optimally with DARE-Stark, a modified version of DARE using STARK proofs that reaches $O(nL + n^2 poly(\kappa))$ bits and maintains optimal $O(n)$ latency. We hope DARE-Stark motivates research into more efficient STARK schemes in the future, which currently have large hidden constants affecting their practical use.

References

1. Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.
2. Ittai Abraham and Gilad Asharov. Gradecast in synchrony and reliable broadcast in asynchrony with optimal resilience, efficiency, and unconditional security. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 392–398. ACM, 2022. doi:10.1145/3519270.3538451.
3. Ittai Abraham, T-H. Hubert Chan, Danny Dolev, Kartik Nayak, Rafael Pass, Ling Ren, and Elaine Shi. Communication Complexity of Byzantine Agreement, Revisited. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, PODC '19, pages 317–326, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3293611.3331629.
4. Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solida: A blockchain protocol based on reconfigurable byzantine consensus. *arXiv preprint arXiv:1612.02916*, 2016.
5. Ittai Abraham, Dahlia Malkhi, Kartik Nayak, Ling Ren, and Alexander Spiegelman. Solidus: An incentive-compatible cryptocurrency based on permissionless byzantine consensus. *CoRR, abs/1612.02916*, 2016.

6 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing (PODC)*, pages 337–346, 2019.

7 Ittai Abraham, Dahlia Malkhi, and Alexander Spiegelman. Asymptotically Optimal Validated Asynchronous Byzantine Agreement. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 337–346, 2019.

8 Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

9 Nicolas Alhaddad, Sourav Das, Sisi Duan, Ling Ren, Mayank Varia, Zhuolun Xiang, and Haibin Zhang. Brief announcement: Asynchronous verifiable information dispersal with near-optimal communication. In Alessia Milani and Philipp Woelfel, editors, *PODC '22: ACM Symposium on Principles of Distributed Computing, Salerno, Italy, July 25 - 29, 2022*, pages 418–420. ACM, 2022. doi:10.1145/3519270.3538476.

10 Nicolas Alhaddad, Sisi Duan, Mayank Varia, and Haibin Zhang. Succinct Erasure Coding Proof Systems. *Cryptology ePrint Archive*, 2021.

11 Yair Amir, Claudiu Danilov, Jonathan Kirsch, John Lane, Danny Dolev, Cristina Nita-Rotaru, Josh Olsen, and David Zage. Scaling byzantine fault-tolerant replication towide area networks. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 105–114. IEEE, 2006.

12 Elli Androulaki, Christian Cachin, Dan Dobre, and Marko Vukolic. Erasure-coded byzantine storage with separate metadata. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*, volume 8878 of *Lecture Notes in Computer Science*, pages 76–90. Springer, 2014. doi:10.1007/978-3-319-14472-6_6.

13 Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73, 1993.

14 Michael Ben-Or, Ran Canetti, and Oded Goldreich. Asynchronous Secure Computation. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 52–61, 1993.

15 Michael Ben-Or, Boaz Kelmer, and Tal Rabin. Asynchronous Secure Computations with Optimal Resilience. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 183–192, 1994.

16 Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. *Cryptology ePrint Archive*, 2018.

17 Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State Machine Replication for the Masses with BFT-SMART. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 355–362. IEEE, 2014.

18 Amey Bhangale, Chen-Da Liu-Zhang, Julian Loss, and Kartik Nayak. Efficient adaptively-secure byzantine agreement for long messages. In Shweta Agrawal and Dongdai Lin, editors, *Advances in Cryptology - ASIACRYPT 2022 - 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5-9, 2022, Proceedings, Part I*, volume 13791 of *Lecture Notes in Computer Science*, pages 504–525. Springer, 2022. doi:10.1007/978-3-031-22963-3_17.

19 Richard E Blahut. *Theory and practice of error control codes*, volume 126. Addison-Wesley Reading, 1983.

20 Manuel Bravo, Gregory Chockler, and Alexey Gotsman. Making Byzantine Consensus Live. In *34th International Symposium on Distributed Computing (DISC)*, volume 179, pages 1–17, 2020.

21 Manuel Bravo, Gregory V. Chockler, and Alexey Gotsman. Making byzantine consensus live. *Distributed Comput.*, 35(6):503–532, 2022. doi:10.1007/s00446-022-00432-y.

22 Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains.* PhD thesis, University of Guelph, 2016.

23 Ethan Buchman, Jae Kwon, and Zarko Milosevic. The latest gossip on BFT consensus. *arXiv preprint arXiv:1807.04938*, pages 1–14, 2018. arXiv:1807.04938.

24 Christian Cachin, Rachid Guerraoui, and Luís E. T. Rodrigues. *Introduction to Reliable and Secure Distributed Programming (2. ed.).* Springer, 2011. doi:10.1007/978-3-642-15260-3.

25 Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. Secure and Efficient Asynchronous Broadcast Protocols. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 524–541. Springer, 2001. doi:10.1007/3-540-44647-8_31.

26 Christian Cachin and Stefano Tessaro. Asynchronous Verifiable Information Dispersal. In *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, pages 191–201. IEEE, 2005.

27 Martina Camaioni, Rachid Guerraoui, Matteo Monti, Pierre-Louis Roman, Manuel Vidigueira, and Gauthier Voron. Chop chop: Byzantine atomic broadcast to the network limit. *arXiv preprint arXiv:2304.07081*, 2023.

28 Jan Camenisch, Manu Drijvers, Timo Hanke, Yvonne Anne Pignolet, Victor Shoup, and Dominic Williams. Internet Computer Consensus. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, 2021:81–91, 2022. doi:10.1145/3519270.3538430.

29 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

30 Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

31 Dario Catalano and Dario Fiore. Vector Commitments and Their Applications. In *Public-Key Cryptography–PKC 2013: 16th International Conference on Practice and Theory in Public-Key Cryptography, Nara, Japan, February 26–March 1, 2013. Proceedings 16*, pages 55–72. Springer, 2013.

32 Jing Chen and Silvio Micali. Algorand. *arXiv preprint arXiv:1607.01341*, 2016.

33 Jinyuan Chen. Fundamental limits of byzantine agreement. *CoRR*, abs/2009.10965, 2020. arXiv:2009.10965.

34 Jinyuan Chen. Optimal error-free multi-valued byzantine agreement. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing, DISC 2021, October 4-8, 2021, Freiburg, Germany (Virtual Conference)*, volume 209 of *LIPIcs*, pages 17:1–17:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.DISC.2021.17.

35 Ashish Choudhury and Arpita Patra. On the communication efficiency of statistically secure asynchronous MPC with optimal resilience. *J. Cryptol.*, 36(2):13, 2023. doi:10.1007/s00145-023-09451-9.

36 Pierre Civit, Muhammad Ayaz Dzulfikar, Seth Gilbert, Vincent Gramoli, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. Byzantine Consensus Is $\Theta(n^2)$: The Dolev-Reischuk Bound Is Tight Even in Partial Synchrony! In Christian Scheideler, editor, *36th International Symposium on Distributed Computing, DISC 2022, October 25-27, 2022, Augusta, Georgia, USA*, volume 246 of *LIPIcs*, pages 14:1–14:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.DISC.2022.14.

37 Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, Matteo Monti, and Manuel Vidigueira. Every Bit Counts in Consensus. *arXiv preprint arXiv:2306.00431*, 2023.

38 Pierre Civit, Seth Gilbert, Rachid Guerraoui, Jovan Komatovic, and Manuel Vidigueira. On the Validity of Consensus. *To appear in PODC 2023*, abs/2301.04920, 2023. doi:10.48550/arXiv.2301.04920.

39 Miguel Correia. From byzantine consensus to blockchain consensus. In *Essentials of Blockchain Technology*, pages 41–80. Chapman and Hall/CRC, 2019.

40 Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. *The Computer Journal*, 49(1):82–96, 2006.

41 Tyler Crain, Vincent Gramoli, Mikel Larrea, and Michel Raynal. Dbft: Efficient leaderless byzantine consensus and its application to blockchains. In *2018 IEEE 17th International Symposium on Network Computing and Applications (NCA)*, pages 1–8. IEEE, 2018.

42 George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. Narwhal and tusk: a dag-based mempool and efficient bft consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 34–50, 2022.

43 Sourav Das, Vinith Krishnan, Irene Miriam Isaac, and Ling Ren. Spurt: Scalable distributed randomness beacon with transparent setup. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2502–2517. IEEE, 2022. doi:10.1109/SP46214.2022.9833580.

44 Sourav Das, Zhuolun Xiang, and Ling Ren. Asynchronous Data Dissemination and its Applications. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2721, 2021.

45 Sourav Das, Thomas Yurek, Zhuolun Xiang, Andrew Miller, Lefteris Kokoris-Kogias, and Ling Ren. Practical asynchronous distributed key generation. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 2518–2534. IEEE, 2022. doi:10.1109/SP46214.2022.9833584.

46 Christian Decker, Jochen Seidel, and Roger Wattenhofer. Bitcoin Meets Strong Consistency. In *Proceedings of the 17th International Conference on Distributed Computing and Networking*, pages 1–10, 2016.

47 Dan Dobre, Ghassan O. Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolic. Proofs of writing for robust storage. *IEEE Trans. Parallel Distributed Syst.*, 30(11):2547–2566, 2019. doi:10.1109/TPDS.2019.2919285.

48 Danny Dolev and Rüdiger Reischuk. Bounds on information exchange for Byzantine agreement. *Journal of the ACM (JACM)*, 1985.

49 Assia Doudou and André Schiper. Muteness Detectors for Consensus with Byzantine Processes. In Brian A. Coan and Yehuda Afek, editors, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing, PODC '98, Puerto Vallarta, Mexico, June 28 - July 2, 1998*, page 315. ACM, 1998. doi:10.1145/277697.277772.

50 Sisi Duan, Xin Wang, and Haibin Zhang. Practical Signature-Free Asynchronous Common Subset in Constant Time. *Cryptology ePrint Archive*, 2023.

51 Sisi Duan, Haibin Zhang, and Boxin Zhao. Waterbear: Information-theoretic asynchronous BFT made practical. *IACR Cryptol. ePrint Arch.*, page 21, 2022. URL: https://eprint.iacr.org/2022/021.

52 Cynthia Dwork, Lynch Nancy, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

53 Facebook. Winterfell: A STARK prover and verifier for arbitrary computations. URL: https://github.com/facebook/winterfell#Performance.

54 Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the Association for Computing Machinery,*, 32(2):374–382, 1985.

55 Adam Gągol, Damian Leśniak, Damian Straszak, and Michał Świętek. ALEPH: Efficient atomic broadcast in asynchronous networks with Byzantine nodes. *AFT 2019 - Proceedings of the 1st ACM Conference on Advances in Financial Technologies*, pages 214–228, 2019. doi:10.1145/3318041.3355467.

56 Chaya Ganesh and Arpita Patra. Broadcast extensions with optimal communication and round complexity. In George Giakkoupis, editor, *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 371–380. ACM, 2016. doi:10.1145/2933057.2933082.

57 Chaya Ganesh and Arpita Patra. Optimal extension protocols for byzantine broadcast and agreement. *IACR Cryptol. ePrint Arch.*, page 63, 2017. URL: `http://eprint.iacr.org/2017/063`.

58 Chaya Ganesh and Arpita Patra. Optimal extension protocols for byzantine broadcast and agreement. *Distributed Comput.*, 34(1):59–77, 2021. `doi:10.1007/s00446-020-00384-1`.

59 James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Low-overhead byzantine fault-tolerant storage. In Thomas C. Bressoud and M. Frans Kaashoek, editors, *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 73–86. ACM, 2007. `doi:10.1145/1294261.1294269`.

60 James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Verifying distributed erasure-coded data. In Indranil Gupta and Roger Wattenhofer, editors, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC 2007, Portland, Oregon, USA, August 12-15, 2007*, pages 139–146. ACM, 2007. `doi:10.1145/1281100.1281122`.

61 Ioannis Kaklamanis, Lei Yang, and Mohammad Alizadeh. Poster: Coded broadcast for scalable leader-based bft consensus. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 3375–3377, 2022.

62 Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. All you need is DAG. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 165–175. ACM, 2021. `doi:10.1145/3465084.3467905`.

63 Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.

64 Ramakrishna Kotla and Michael Dahlin. High throughput byzantine fault tolerance. In *International Conference on Dependable Systems and Networks, 2004*, pages 575–584. IEEE, 2004.

65 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

66 Andrew Lewis-Pye. Quadratic worst-case message complexity for State Machine Replication in the partial synchrony model, 2022. `doi:10.48550/ARXIV.2201.01107`.

67 Fan Li and Jinyuan Chen. Communication-efficient signature-free asynchronous byzantine agreement. In *IEEE International Symposium on Information Theory, ISIT 2021, Melbourne, Australia, July 12-20, 2021*, pages 2864–2869. IEEE, 2021. `doi:10.1109/ISIT45174.2021.9518010`.

68 Benoît Libert, Marc Joye, and Moti Yung. Born and Raised Distributively: Fully Distributed Non-Interactive Adaptively-Secure Threshold Signatures with Short Shares. *Theoretical Computer Science*, 645:1–24, 2016.

69 Yuan Lu, Zhenliang Lu, Qiang Tang, and Guiling Wang. Dumbo-MVBA: Optimal Multi-Valued Validated Asynchronous Byzantine Agreement, Revisited. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 129–138, 2020.

70 Loi Luu, Viswesh Narayanan, Kunal Baweja, Chaodong Zheng, Seth Gilbert, and Prateek Saxena. Scp: A computationally-scalable byzantine consensus protocol for blockchains. *Cryptology ePrint Archive*, 2015.

71 Dahlia Malkhi, Kartik Nayak, and Ling Ren. Flexible byzantine fault tolerance. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, pages 1041–1053, 2019.

72 Ralph C. Merkle. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology — CRYPTO*, 1987.

73 Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

74 Atsuki Momose and Ling Ren. Multi-threshold byzantine fault tolerance. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1686–1699, 2021.

75 Atsuki Momose and Ling Ren. Optimal Communication Complexity of Authenticated Byzantine Agreement. In *35th International Symposium on Distributed Computing (DISC)*, volume 209, pages 32:1–32:0. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany, 2021.

76 Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-Free Asynchronous Binary Byzantine Consensus with t < n/3, O(n2) Messages, and O(1) Expected Time. *J. ACM*, 62(4):31:1–31:21, 2015. doi:10.1145/2785953.

77 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, page 21260, 2008.

78 Oded Naor and Idit Keidar. Expected Linear Round Synchronization: The Missing Link for Linear Byzantine SMR. *34th International Symposium on Distributed Computing (DISC)*, 179, 2020.

79 Kartik Nayak, Ling Ren, Elaine Shi, Nitin H. Vaidya, and Zhuolun Xiang. Improved extension protocols for byzantine broadcast and agreement. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 28:1–28:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.DISC.2020.28.

80 Nuno Ferreira Neves, Miguel Correia, and Paulo Verissimo. Solving Vector Consensus with a Wormhole. *IEEE Transactions on Parallel and Distributed Systems*, 16(12):1120–1131, 2005.

81 Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

82 Irving S Reed and Gustave Solomon. Polynomial odes over certain finite fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

83 Victor Shoup. Practical Threshold Signatures. In Bart Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 207–220. Springer, 2000. doi:10.1007/3-540-45539-6_15.

84 Victor Shoup and Nigel P Smart. Lightweight asynchronous verifiable secret sharing with optimal resilience. *Cryptology ePrint Archive*, 2023.

85 Alexander Spiegelman. In Search for an Optimal Authenticated Byzantine Agreement. In Seth Gilbert, editor, *35th International Symposium on Distributed Computing (DISC 2021)*, volume 209 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 38:1–38:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.DISC.2021.38.

86 Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. Bullshark: Dag bft protocols made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2705–2718, 2022.

87 Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Verissimo. Efficient byzantine fault-tolerance. *IEEE Transactions on Computers*, 62(1):16–30, 2011.

88 Gavin Wood et al. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151(2014):1–32, 2014.

89 Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. HotStuff: BFT Consensus with Linearity and Responsiveness. *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, pages 347–356, 2019.

90 Thomas Yurek, Licheng Luo, Jaiden Fairoze, Aniket Kate, and Andrew Miller. hbacss: How to robustly share many secrets. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022*. The Internet Society, 2022. URL: https://www.ndss-symposium.org/ndss-paper/auto-draft-245/.

A Sync: Implementation

In this section, we provide the remaining pseudocode of DISPERSER, namely the SYNC component. For a complete formal proof of the correctness and complexity of DISPERSER, please refer to the extended version of this paper. Throughout the entire section, $X = Y = \sqrt{n}$. For simplicity, we assume that $\sqrt{n}$ is an integer.

Algorithm description. The pseudocode of SYNC is given in Algorithm 4, and it highly resembles RARESYNC [36]. Each process P_i has an access to two timers: (1) $view\_timer_i$, and (2) $dissemination\_timer_i$. A timer exposes the following interface:

- measure(Time x): After exactly x time as measured by the local clock, the timer expires (i.e., an expiration event is received by the host). As local clocks can drift before GST, timers might not be precise before GST: x time as measured by the local clock may not amount to x real time.
- cancel(): All previously invoked measure($\cdot$) methods (on that timer) are cancelled. Namely, all pending expiration events (associated with that timer) are removed from the event queue.

We now explain how SYNC works, and we do so from the perspective of a correct process P_i. When P_i starts executing SYNC (line 11), it starts measuring $view\_duration = \Delta + 2\delta$ time using $view\_timer_i$ (line 12) and enters the first view (line 13).

Once $view\_timer_i$ expires (line 14), which signals that P_i's current view has finished, P_i notifies every process of this via a VIEW-COMPLETED message (line 15). When P_i learns that $2t + 1$ processes have completed some view $V \geq view_i$ (line 16) or any process started some view $V' > view_i$ (line 22), P_i prepares to enter a new view (either $V + 1$ or V'). Namely, P_i (1) cancels $view\_timer_i$ (line 19 or line 25), (2) cancels $dissemination\_timer_i$ (line 20 or line 26), and (3) starts measuring δ time using $dissemination\_timer_i$ (line 21 or line 27). Importantly, P_i measures δ time (using $dissemination\_timer_i$) before entering a new view in order to ensure that P_i enters only $O(1)$ views during the time period $[\text{GST}, \text{GST} + 3\delta)$. Finally, once $dissemination\_timer_i$ expires (line 28), P_i enters a new view (line 31).

B STARKs

First introduced in [16], STARKs are succinct, universal, transparent arguments of knowledge. For any function f (computable in polynomial time) and any (polynomially-sized) y, a STARK can be used to prove the knowledge of some x such that $f(x) = y$. Remarkably, the size of a STARK proof is $O(poly(\kappa))$. At a very high level, a STARK proof is produced as follows: (1) the computation of $f(x)$ is unfolded on an execution trace; (2) the execution trace is (RS) over-sampled for error amplification; (3) the correct computation of f is expressed as a set of algebraic constraints over the trace symbols; (4) the trace symbols are organized in a Merkle tree [72]; (5) the tree's root is used as a seed to pseudo-randomly sample the trace symbols. The resulting collection of Merkle proofs proves that, for some known (but not revealed) x, $f(x) \neq y$ only with cryptographically low probability (negligible in κ). STARKs are non-interactive, require no trusted setup (they are transparent), and their security reduces to that of cryptographic hashes in the Random Oracle Model (ROM) [13].

C Further Analysis of DARE

In this section, we provide a brief good-case analysis of DARE and discuss how DARE can be adapted to a model with unknown δ.

Algorithm 4 SYNC: Pseudocode (for process P_i).

Uses:
  Timer $view\_timer_i$
  Timer $dissemination\_timer_i$

Functions:
  leaders(View V) $\equiv \{P_{\left((V \bmod \sqrt{n})-1\right)\sqrt{n}+1}, P_{\left((V \bmod \sqrt{n})-1\right)\sqrt{n}+2}, ..., P_{\left((V \bmod \sqrt{n})-1\right)\sqrt{n}+\sqrt{n}}\}$

Constants:
  Time $view\_duration = \Delta + 2\delta = \delta\sqrt{n} + 3\delta + 2\delta$

Variables:
  View $view_i \leftarrow 1$
  T_Signature $view\_sig_i \leftarrow \bot$

upon init: ▷ start of the algorithm
  $view\_timer_i$.measure($view\_duration$)
  trigger advance(1)

upon $view\_timer_i$ **expires:**
  broadcast ⟨VIEW-COMPLETED, $view_i$, share_sign$_i$($view_i$)⟩

upon exists View $V \geq view_i$ with $2t+1$ ⟨VIEW-COMPLETED, V, P_Signature sig⟩ received messages:
  $view\_sig_i \leftarrow$ combine$\big(\{sig \mid sig$ is received in the VIEW-COMPLETED messages$\}\big)$
  $view_i \leftarrow V + 1$
  $view\_timer_i$.cancel()
  $dissemination\_timer_i$.cancel()
  $dissemination\_timer_i$.measure(δ)

upon reception of ⟨ENTER-VIEW, View V, T_Signature sig⟩ with $V > view_i$:
  $view\_sig_i \leftarrow sig$
  $view_i \leftarrow V$
  $view\_timer_i$.cancel()
  $dissemination\_timer_i$.cancel()
  $dissemination\_timer_i$.measure(δ)

upon $dissemination\_timer_i$ **expires:**
  broadcast ⟨ENTER-VIEW, $view_i$, $view\_sig_i$⟩
  $view\_timer_i$.measure($view\_duration$)
  trigger advance($view_i$)

C.1 Good-Case Complexity

For the good-case complexity, we consider only executions where GST $= 0$ and where all processes behave correctly. This is sometimes also regarded as the common case since, in practice, there are usually no failures and the network behaves synchronously. Throughout the entire subsection, $X = Y = \sqrt{n}$.

In such a scenario, the good-case bit complexity of DARE is $O(n^{1.5}L + n^2\kappa)$. As all processes are correct and synchronized at the starting view, DISPERSER terminates after only one view. The $n^{1.5}L$ term comes from this view: the first $\sqrt{n}$ correct leaders broadcast their full L-bit proposal to all other processes. The $n^{2.5}\kappa$ term is reduced to only $n^2\kappa$ (only the CONFIRM messages sent by correct processes at line 29) since DISPERSER terminates after just one view.

The good-case latency of DARE is essentially the sum of the good-case latencies of the Dispersal, Agreement, and Retrieval phases:

$$\underbrace{O(\sqrt{n}\cdot\delta)}_{\text{DISPERSAL}} + \underbrace{O(\delta)}_{\text{AGREEMENT}} + \underbrace{O(\delta)}_{\text{RETRIEVAL}} = O(\sqrt{n}\cdot\delta).$$

Thus, the good-case latency of DARE is $O(\sqrt{n}\cdot\delta)$.

C.2 DARE (and DARE-Stark) with Unknown δ

To accommodate for unknown δ, two modifications to DARE are required:

- DISPERSER must accommodate for unknown δ. We can achieve this by having SYNC increase the ensured overlap with every new view (by increasing *view_duration* for every new view).
- AGREEMENT must accommodate for unknown δ. Using the same strategy as for SYNC, AGREEMENT can tolerate unknown δ. (The same modification makes DARE-STARK resilient to unknown δ.)