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 \( \kappa \) 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 \( O(n^{1.5}L + n^2\kappa) \) 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 \( O(nL + n^2 \text{poly}(\kappa)) \). Both DARE and DARE-STARK achieve optimal \( O(n) \) worst-case latency.

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 $\text{valid}(v) = \text{true}$.

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

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 $\kappa$ 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 [23], 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 \(O(nL + n^2 \text{poly}(\kappa))\) 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 \(\tilde{O}(n s)\) 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 \(o(n^2 L)\) 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.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Model</th>
<th>Cryptography</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC [25](^{†})</td>
<td>Async</td>
<td>PKI, TS [68]</td>
<td>(O(n^4 L + n^2 \kappa + n^3)) (O(1))</td>
</tr>
<tr>
<td>VABA [6]</td>
<td>Async</td>
<td>above</td>
<td>(O(n^2 L + n^2 \kappa)) (O(1))</td>
</tr>
<tr>
<td>Dumbo-MVBA [69]</td>
<td>Async</td>
<td>above + ECC</td>
<td>(O(nL + n^3 \kappa)) (O(1))</td>
</tr>
<tr>
<td>PBFT [30, 17]</td>
<td>PSync</td>
<td>PKI</td>
<td>(O(n^3 L + n^4 \kappa)) (O(n))</td>
</tr>
<tr>
<td>HotStuff [89]</td>
<td>PSync</td>
<td>above + TS</td>
<td>(O(n^3 L + n^3 \kappa)) (O(n))</td>
</tr>
<tr>
<td>Quad [36, 66]</td>
<td>PSync</td>
<td>above</td>
<td>(O(n^2 L + n^2 \kappa)) (O(n))</td>
</tr>
<tr>
<td>DARE</td>
<td>PSync</td>
<td>above + ECC</td>
<td>(O(n^{1.5} L + n^{2.5} \kappa)) (O(n))</td>
</tr>
<tr>
<td>DARE-Stark</td>
<td>PSync</td>
<td>above + STARK</td>
<td>(O(nL + n^3 \kappa)) (O(n))</td>
</tr>
</tbody>
</table>

Figure 1: Performance of various consensus algorithms with \(L\)-bit values and \(\kappa\)-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 \(O(n^2 L)\) 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 + \(\delta\) due to asynchrony (\(\delta\) is the maximum message delay after GST). Once GST is reached, all \(t\) processes wake up and broadcast their value, for a total of \(O(tnL) = O(n^2 L)\) 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 valid($v$) = 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 $\delta$ time (i.e., broadcast to the first subgroup, wait $\delta$ 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 ($O(\text{poly}(\kappa))$ bits) of a statement on some value without having to share that value. As an example, a process $P_i$ can send $⟨h, \sigma_{\text{STARK}}⟩$ to a process $P_j$, which can use $\sigma_{\text{STARK}}$ to verify the statement “$\exists v : \text{valid}(v) = \text{true} \land \text{hash}(v) = h$”, 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 $O(\text{poly}(\kappa))$ bit-sized messages, obtaining DARE-Stark. This yields the overall near-optimal bit complexity of $O(nL + n^2 \text{poly}(\kappa))$. Currently, the main drawback of STARKs is their size and computation time in practice, 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.²

### 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 \(O(n^2L + n^3\kappa)\) 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 \(O(nL + n^3\kappa)\) 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 \(O(nL + n^3\kappa)\). Interestingly, it has been demonstrated that synchronous Byzantine agreement can be implemented with a bit complexity of \(O(n(L + \text{poly}(\kappa)))\) using randomization [18]. The Dolev-Reischuk bound [48] is not violated in this case since the implementation tolerates a negligible (with \(\kappa\)) 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 \(O(nL + n^3\kappa)\).

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 \( O(n^2 L) \). 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 \( \Omega(n^2 L) \).

**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 \( \text{Process} = \{P_1, P_2, ..., P_n\} \) 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 \( \delta \) such that the message delays are bounded by \( \delta \) after GST. We assume that \( \delta \) 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 \( \delta \) is unknown (these modifications are specified in Appendix C.2), although their complexity might be higher.

Cryptographic primitives. Throughout the paper, hash(\( \cdot \)) denotes a collision-resistant hash function. The codomain of the aforementioned hash(\( \cdot \)) function is denoted by 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 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 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 \( \Sigma_m \), verify_sig(\( m, \Sigma_m \)) returns true if and only if \( \Sigma_m \) is a valid signature for \( m \). Where appropriate, the verifications are left implicit. We denote by P_Signature and T_Signature the set of partial and threshold signatures, respectively. The size of cryptographic objects (i.e., hashes, signatures) is denoted by \( \kappa \); we assume that \( \kappa > \log n \).

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 encode(\( \cdot \)) and decode(\( \cdot \)) to denote RS’ encoding and decoding algorithms. In a nutshell, 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. 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 Consensus be a partially synchronous Byzantine consensus algorithm, and let \( \mathcal{E}(\text{Consensus}) \) denote the set of all possible executions. Let \( \alpha \in \mathcal{E}(\text{Consensus}) \) be an execution, and \( t_d(\alpha) \) be the first time by which all correct processes have decided in \( \alpha \). The bit complexity of \( \alpha \) is the total number of bits sent by correct processes during the time period [GST, \( \infty \)]. The latency of \( \alpha \) is \( \max(0, t_d(\alpha) - \text{GST}) \).

The bit complexity of Consensus is defined as

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

\footnote{For \( \kappa \leq \log n \), \( t \in O(n) \) faulty processes would have computational power exponential in \( \kappa \), breaking cryptographic hardness assumptions.}
Similarly, the latency of Consensus is defined as
\[ \max_{\alpha \in \mathcal{E}(\text{Consensus})} \{ \text{latency of } \alpha \} . \]

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

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 \((h, \Sigma_h)\), then \text{verify\_sig}(h, \Sigma_h) = true.
- **Termination:** Every correct process eventually acquires at least one hash-signature pair.
- **Redundancy:** Let a correct process obtain a threshold signature \(\Sigma_h\) such that \text{verify\_sig}(h, \Sigma_h) = true, for some hash value \(h\). Then, (at least) \(t + 1\) correct processes have obtained a value \(v\) such that (1) \text{hash}(v) = h, and (2) \text{valid}(v) = 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. In Agreement, processes propose and decide pairs \((h \in \text{Hash Value}, \Sigma_h \in \text{T Signature})\); moreover, \(\text{valid}(h, \Sigma_h) \equiv \text{verify sig}(h, \Sigma_h)\).

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) \(\bot\). Eventually, all correct processes output the same value. Formally, Retriever exposes the following interface:

- **request input** \((v \in \text{Value} \cup \{\bot\})\) a process inputs a value or \(\bot\); each correct process invokes input(·) exactly once. Moreover, the following is assumed:
  - No two correct processes invoke input\((v_1 \in \text{Value})\) and input\((v_2 \in \text{Value})\) with \(v_1 \neq v_2\).
  - At least \(t+1\) correct processes invoke input\((v \in \text{Value})\) (i.e., \(v \neq \bot\)).

- **indication output** \((v' \in \text{Value})\): 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 \(v \neq \bot\) into \(n\) symbols. Each correct process \(Q\) which inputs \(v \neq \bot\) to Retriever encodes \(v\) into \(n\) RS symbols \(s_1, s_2, \ldots, s_n\). \(Q\) 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.
We start by proving the correctness of DARE. Algorithm 1 gives DARE’s pseudocode (for process \( P_i \)).

```plaintext
Algorithm 1 DARE: Pseudocode (for process \( P_i \)).

1: Uses:
2: \( \triangleright \) bits: \( O(n^{1.5}L + n^2\log n) \), latency: \( O(n) \) (see §5)
3: Disperser, instance disperser
4: \( \triangleright \) bits: \( O(n^2\kappa) \), latency: \( O(n) \) (see [36])
5: Agreement, instance agreement
6: \( \triangleright \) bits: \( O(nL + n^2\log n) \), latency: \( O(1) \) (see [44])
7: Retriever, instance retriever
8: upon \( \text{propose}(v_i \in \text{Value}) \):
9: \hspace{1em} \text{invoke disperser.disperse}(v_i)
10: upon disperser.acquire\((h_i \in \text{Hash.Value}, \Sigma_i \in \text{T.Signature})\):
11: \hspace{1em} \text{invoke agreement.propose}(h_i, \Sigma_i)
12: upon agreement.decide\((h \in \text{Hash.Value}, \Sigma_h \in \text{T.Signature})\):
13: \hspace{1.5em} \( v \leftarrow \) an obtained value such that \( \text{hash}(v) = h \) (if such a value was not obtained, \( v = \bot \))
14: \hspace{1em} \text{invoke retriever.input}(v)
15: upon retriever.output\((\text{Value} \; v')\):
16: \hspace{1.5em} \text{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 \text{Disperser} (line 9). Eventually, \( P_i \) acquires a hash-signature pair \((h_i, \Sigma_i)\) (line 10) due to the termination property of \text{Disperser}.
2. **Agreement**: Process \( P_i \) proposes the previously acquired hash-signature pair \((h_i, \Sigma_i)\) to \text{Agreement} (line 11). As \text{Agreement} satisfies termination and agreement, all correct processes eventually agree on a hash-signature pair \((h, \Sigma_h)\) (line 12).
3. **Retrieval**: Once process \( P_i \) decides \((h, \Sigma_h)\) from \text{Agreement}, it checks whether it has previously obtained a value \( v \) with \( \text{hash}(v) = h \) (line 13). If it has, \( P_i \) inputs \( v \) to \text{Retriever}; otherwise, \( P_i \) inputs \( \bot \) (line 14). The required preconditions for \text{Retriever} are met:
   - No two correct processes input different non-\( \bot \) values to \text{Retriever} as \text{hash}(\cdot) is collision-resistant.
   - At least \((t + 1)\) correct processes input a value (and not \( \bot \)) to \text{Retriever}. Indeed, as \( \Sigma_h \) is obtained by a correct process, \( t + 1 \) correct processes have obtained a value \( v \neq \bot \) with \( \text{hash}(v) = h \) (due to redundancy of \text{Disperser}), and all of these processes input \( v \).

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

4.3 Proof of Correctness & Complexity

We start by proving the correctness of DARE.

\( \triangleright \) **Theorem 1.** DARE is correct.

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

As \((h, \Sigma_h)\) is decided by all correct processes, at least \(t + 1\) correct processes \(P_i\) have obtained a value \(v\) such that (1) \(\text{hash}(v) = h\), and (2) \(\text{valid}(v) = \text{true}\) (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 \(\text{hash}(\cdot)\) 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.

\[\textbf{Theorem 2. DARE achieves } O(n^{1.5}L + n^{2.5}\kappa) \text{ bit complexity and } O(n) \text{ latency.} \]

\[\textbf{Proof.} \text{ As DARE is a sequential composition of its building blocks, its complexity is the sum of the complexities of (1) Disperser, (2) Agreement, and (3) Retriever. Hence, the bit complexity is} \]

\[O(n^{1.5}L + n^{2.5}\kappa) + O(n^2\kappa) + O(nL + n^2\log n) = O(n^{1.5}L + n^{2.5}\kappa). \]

Similarly, the latency is \(O(n)\).

\[\]

\section{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.

\subsection{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 \(V\), \(\text{leaders}(V)\) 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 \(\delta\)-waiting step in between (line 15). Before we thoroughly explain the pseudocode, we introduce Sync, Disperser’s view synchronization [36, 89, 66] algorithm.

\textbf{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:

\[
\text{indication advance}(V \in \text{View}): \text{ a process enters a new view } V.
\]

Sync guarantees eventual synchronization: there exists a time \(\tau_{\text{sync}} \geq \text{GST} \text{ (synchronization time)}\) such that (1) all correct processes are in the same view \(V_{\text{sync}} \) (synchronization view) from time \(\tau_{\text{sync}}\) to (at least) time \(\tau_{\text{sync}} + \Delta\), and (2) \(V_{\text{sync}}\) has a correct leader. We denote by \(V_{\text{sync}}^*\) the smallest synchronization view, whereas \(\tau_{\text{sync}}^*\) denotes the first synchronization time. Similarly, \(V_{\text{max}}\) denotes the greatest view entered by a correct process before GST.\(^{\ref{footnote:define-max-view}}\)

\(^{\ref{footnote:define-max-view}}\) When such a view does not exist, \(V_{\text{max}} = 0\)
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 $O(\Delta) = O(\frac{n}{\sqrt{X}})$ 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 $\frac{\kappa}{X}$ consecutive views. As $O(\frac{n}{\sqrt{X}})$ views (after GST) are required to reach a correct leader, Sync exchanges $O(\frac{n}{\sqrt{X}}) \cdot O(n^2\kappa) = O(\frac{n^3\kappa}{X})$ bits (before synchronization, i.e., before $\tau_{sync} + \Delta$); since each view takes $O(\frac{n}{\sqrt{X}})$ time, synchronization is ensured within $O(\frac{n}{\sqrt{X}}) \cdot O(\frac{n}{\sqrt{X}}) = O(\frac{n^2}{X})$ 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 GST $+ 3\delta$.
- **Limited entrance:** In the time period [GST, 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) $\Delta$ time.
- **Limited synchronization view:** $V_{sync}^* - V_{max} = O(\frac{n}{\sqrt{X}})$.
- **Complexity:** Sync exchanges $O(\frac{n^2}{X})$ bits during the time period [GST, $\tau_{sync}^* + \Delta$], and it synchronizes all correct processes within $O(\frac{n^2}{X})$ time after GST ($\tau_{sync}^* + \Delta - \text{GST} = O(\frac{n^2}{X})$).

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 **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 **DISPERAL** messages. As already mentioned, $P_l$ sends its proposal to $Y$-sized groups of processes (line 14) with a $\delta$-waiting step in between (line 15). When a correct (non-leader) process $P_i$ (which participates in the view $V$) receives a **DISPERAL** 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 **DISPERAL** (line 30).

**5.2 Analysis.** Once all correct processes synchronize in the view $V_{sync}^*$ (the smallest synchronization view), all correct processes acquire a hash-signature pair. Indeed, $\Delta = \delta \frac{n}{\sqrt{X}} + 3\delta$ time is sufficient for a correct leader $P_l \in \text{leaders}(V_{sync}^*)$ to (1) disperse its proposal $\text{proposal}_l$ to all processes (line 14), (2) collect $2t + 1$ partial signatures for $h = \text{hash}(\text{proposal}_l)$ (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 **DISPERAL** (line 30).

**DISC 2023**
Every Bit Counts in Consensus

**Algorithm 2** DISPERSER: Pseudocode (for process $P_i$).

1: Uses:
2: ```
SYNC, instance $sync$ \hspace{1cm} \triangleright$ ensures a $\Delta = \delta + 3\delta$ overlap in a view with a correct leader
3: upon init:
4: ```
5: ```
6: ```
7: ```
8: ```
9: ```
10: ```
11: ```
12: ```
13: ```
14: ```
15: ```
16: ```
17: ```
18: ```
19: ```
20: ```
21: ```
22: ```
23: ```
24: ```
25: ```
26: ```
27: ```
28: ```
29: ```
30: ```

**Complexity.** DISPERSER terminates once all correct processes are synchronized in a view with a correct leader. The synchronization is ensured in $O\left(\frac{n^2}{\Delta} \log Y\right)$ time after GST (as $\tau_{sync}^* + \Delta - GST = O\left(\frac{n^2}{\Delta} \log Y\right)$). Hence, DISPERSER terminates in $O\left(\frac{n^2}{\Delta} \log Y\right)$ 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 DISPERSER, ACK and CONFIRM message in the aforementioned periods:

- Unsynchronized period: Due to the $\delta$-waiting step (line 15), each correct process sends DISPERSER 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 DISPERSER 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 $\kappa$ bits, all correct processes send

$$n \cdot \left( O(YL) + O(X\kappa) + O(n\kappa) \right)$$

$$= O(nYL + n^2\kappa)$$

bits via DISPERSER, ACK and CONFIRM messages.
Synchronized period: Recall that all correct processes acquire a hash-signature pair (and stop executing Disperser) by time \( \tau_{\text{sync}}^* + \Delta \), and they do so in the view \( V_{\text{sync}}^* \). As correct processes enter monotonically increasing views, no correct process enters a view greater than \( V_{\text{sync}}^* \).

By the stabilization property of Sync, each correct process enters a view \( V \geq V_{\text{max}} \) by time GST + 3\( \delta \). Moreover, until \( \tau_{\text{sync}}^* + \Delta \), 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 \( V_{\text{max}} < V < V_{\text{sync}}^* \); otherwise, \( V = V_{\text{sync}}^* \) as processes overlap for \( \Delta \) time in \( V \) (due to the overlapping property of Sync).

Hence, for each view \( V \) with \( V_{\text{max}} < V < V_{\text{sync}}^* \), all correct processes send \( O(n\kappa) \) bits (all through ACK messages; line 20). In \( V_{\text{max}} \) and \( V_{\text{sync}}^* \), all correct processes send (1) \( 2 \cdot O(XnL) \) bits through Dispersal messages (line 14), (2) \( 2 \cdot O(n\kappa) \) bits through ACK messages (line 20), and (3) \( 2 \cdot O(Xn\kappa) \) bits through CONFINIRM messages (line 25).

Therefore, all correct processes send

\[
O\left(\frac{n}{X}\right) \cdot \frac{O(Xn\kappa)}{V_{\text{sync}}^* - V_{\text{max}}} + \frac{O(XnL)}{\text{ACK in } V_{\text{max}} \text{ and } V_{\text{sync}}^*} + \frac{O(n\kappa)}{\text{ACK in } V_{\text{sync}}^*} + \frac{O(Xn\kappa)}{\text{CONFIRM in } V_{\text{max}} \text{ and } V_{\text{sync}}^*} = O(nXL + n^2\kappa) \text{ bits via Dispersal, ACK and CONFIRM messages.}
\]

We cannot neglect the complexity of Sync, which exchanges \( O\left(\frac{n^2\kappa}{X}\right) \) bits during the time period [GST, \( \tau_{\text{sync}}^* + \Delta \)]. Hence, the total number of bits Disperser exchanges is

\[
O(nYL + n^2\kappa) + O(nXL + n^2\kappa) + O\left(\frac{n^3\kappa}{X}\right) = O(nYL + nXL + n^2\kappa).
\]

With \( X = Y = \sqrt{n} \), Disperser terminates in optimal \( O(n) \) time, and exchanges \( O(n^{1.5}L + n^2\kappa) \) bits. Our analysis is illustrated in Figure 4.

![Figure 4 Illustration of Disperser’s bit complexity.](image)

### 6 DARE-Stark

In this section, we present DARE-Stark, a variant of DARE which relies on STARK proofs. Importantly, DARE-Stark achieves \( O(nL + n^2\text{poly}(\kappa)) \) 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 $h = \text{hash}(v)$. 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 $s_1, \ldots, s_n$; (2) $P$ sends each $s_i$ to $P_t$; (3) upon receiving $t + 1$ identical copies of $s_i$, $P_t$ 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 $\text{shard}_i(\cdot)$ by

$$\text{shard}_i(v \in \text{Value}) = \begin{cases} \langle \text{hash}(v), \text{encode}_i(v) \rangle, & \text{if and only if } \text{valid}(v) = \text{true} \\ \bot, & \text{otherwise,} \end{cases}$$

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

**Theorem 3.** Let $i_1, \ldots, i_{t+1}$ be distinct indices in $[1, n]$. Let $h$ be a hash, let $s_1, \ldots, s_{t+1}$ be RS symbols, let $\text{stark}_1, \ldots, \text{stark}_{t+1}$ be STARK proofs such that, for every $k \in [1, t + 1]$, $\text{stark}_k$ proves knowledge of some (undisclosed) $v_k$ such that $\text{shard}_{i_k}(v_k) = (h, s_k)$. We have that

$$v = \text{decode}(\{s_1, \ldots, s_k\})$$

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

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

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

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

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

1. **Dispersal:** Upon proposing a value $v_i$ (line 9), $P_t$ sends (line 14) to each process $P_k$ (1) $(h_k, s_k) = \text{shard}_i(v_i)$ (computed at line 12), and (2) $\text{stark}_k = \text{proof}_i(v_i)$ (computed at line 13). In doing so (see Theorem 3), $P_t$ proves to $P_k$ that $h_k = \text{hash}(v_i)$ is the hash of a
\[\textbf{Algorithm 3 DARE-STARK: Pseudocode (for process } P_i \text{).}\]

1: \textbf{Uses:}
2: \textbf{AGREEMENT, instance agreement}
3: \textbf{upon} init:
4: \textbf{Map}(\text{Hash}\_\text{Value} \to \langle \text{Symbol, STARK} \rangle) \quad \text{proposal\_shards}_i \leftarrow \text{empty}
5: \textbf{Map}(\text{Hash}\_\text{Value} \to \text{Set(\text{Symbol})}) \quad \text{decision\_symbols}_i \leftarrow \text{empty}
6: \textbf{Bool} decided_i \leftarrow \text{false}
7: \textbf{upon} propose(Value \_i): \quad \text{proposed\_hash}_i \leftarrow \text{hash}(v_i)
8: \quad \text{for} \text{ integer } k \leftarrow 1 \text{ to } n:
9: \quad \text{(Hash } h_k, \text{ Symbol } s_k) \leftarrow \text{shard}_k(v_i)
10: \quad \text{STARK } stark_k \leftarrow \text{proof}_k(v_i)
11: \quad \text{send} \langle \text{DISPERSAL}, h_k, s_k, stark_k \rangle \text{ to } P_k
12: \textbf{upon} reception of \langle \text{DISPERSAL}, \text{Hash}\_\text{Value } h, \text{Symbol } s, \text{STARK } stark \rangle \text{ from process } P_j \text{ and } stark
13: \text{proves} \text{shard}_j((h, s)) = (h, s):
14: \quad \text{proposed\_shards}_j[h] \leftarrow (s, stark)
15: \text{send} \langle \text{ACK, share\_sign}(h) \rangle \text{ to } P_j
16: \textbf{upon} reception of \langle \text{RETRIEVE}, h, s, stark \rangle \text{ from process } P_j \text{ and } stark
17: \quad \text{proves} \text{shard}_j((h, s)) = (h, s):
18: \quad \text{decision\_symbols}_j[h] \leftarrow \text{decision\_symbols}_j[h] \cup \{s\}
19: \text{upon} \langle \text{ACK, } P_i, \text{Signature } sig \rangle \text{ is received from } 2t + 1 \text{ processes:}
20: \quad \text{T\_Signature } \Sigma \leftarrow \text{combine}(\{\text{sig} | \text{sig is received in the ACK messages}\})
21: \quad \text{invoke}\ \text{agreement\_propose}(\text{proposed\_hash}_i, \Sigma)
22: \quad \text{upon}\ \text{agreement\_decide}(\text{Hash}\_\text{Value } h, \text{T\_Signature } \Sigma) \text{ with proposal\_shards}_j[h] \neq \perp:
23: \quad \langle \text{Symbol } s, \text{STARK } stark \rangle \leftarrow \text{proposed\_shards}_j[h]
24: \quad \text{broadcast} \langle \text{RETRIEVE}, h, s, stark \rangle
25: \textbf{upon} reception of \langle \text{RETRIEVE}, \text{Hash}\_\text{Value } h, \text{Symbol } s, \text{STARK } stark \rangle \text{ from process } P_j \text{ and } stark
26: \quad \text{proves} \text{shard}_j((h, s)) = (h, s):
27: \quad \text{decision\_symbols}_j[h] \leftarrow \text{decision\_symbols}_j[h] \cup \{s\}
28: \text{upon } (1) \text{ exists } \text{Hash}\_\text{Value } h \text{ such that } \text{decision\_symbols}_j[h] \text{ has } t + 1 \text{ elements, and } (2) \text{ decided}_i = \text{false:}
29: \quad \text{decided}_i \leftarrow \text{true}
30: \quad \text{trigger}\ \text{decide}(\text{decode}(\text{decision\_symbols}_j[h]))

valid proposal, whose k-th RS symbol is encod$_k$(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 \(\Sigma\) for hash(v$_i$) (line 20), $P_i$ proposes \langle hash(v$_i$), \Sigma \rangle 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 decode(\(S\)) (line 30).

**Analysis.** Upon proposing a value $v_i$ (line 9), a correct process $P_i$ sends shard$_k(v_i)$ and proof$_k(v_i)$ to each process $P_k$ (line 14). Checking proof$_k(v_i)$ against 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 hash(v$_i$)). As $2t + 1$ processes are correct, $P_i$ is guaranteed to eventually gather a $(2t + 1)$-threshold signature \(\Sigma\) for hash(v$_i$) (line 20). Upon doing so, $P_i$ proposes \langle hash(v$_i$), \Sigma \rangle 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 $P_1,\ldots,P_{t+1}$ 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 decode$(S)$ (line 30). By Theorem 3, every correct process eventually decides the same valid value $v$ (with $h = \text{hash}(v)$).

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 $O(\frac{L}{n} + \text{poly}(\kappa))$, (2) $n$ Ack messages, each of size $O(\kappa)$, and (3) $n$ retrieve messages, each of size $O(\frac{L}{n} + \text{poly}(\kappa))$. Therefore, the bit complexity of DARE-Stark is $O(nL + n^2\text{poly}(\kappa))$. 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 ($\kappa$ 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 $O(nL + n^2\text{poly}(\kappa))$ 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


Every Bit Counts in Consensus


Every Bit Counts in Consensus


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) \texttt{view\_timer}_i, and (2) \texttt{dissemination\_timer}_i. A timer exposes the following interface:

- \texttt{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.
- \texttt{cancel}(): All previously invoked \texttt{measure}() 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 $\texttt{view\_duration} = \Delta + 2\delta$ time using \texttt{view\_timer}_i (line 12) and enters the first view (line 13).

Once \texttt{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 \texttt{view\_completed} message (line 15). When $P_i$ learns that $2t + 1$ processes have completed some view $V \geq \texttt{view}_i$ (line 16) or any process started some view $V' > \texttt{view}_i$ (line 22), $P_i$ prepares to enter a new view (either $V + 1$ or $V'$). Namely, $P_i$ (1) cancels \texttt{view\_timer}_i (line 19 or line 25), (2) cancels \texttt{dissemination\_timer}_i (line 20 or line 26), and (3) starts measuring $\delta$ time using \texttt{dissemination\_timer}_i (line 21 or line 27). Importantly, $P_i$ measures $\delta$ time (using \texttt{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 \texttt{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 $O(\text{poly}(\kappa))$. 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 $\kappa$). 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 $\delta$. 
Dispersal, Agreement, and Retrieval phases: just one view.

Confirmation process is correct and synchronized at the starting view, the entire subsection, practice, there are usually no failures and the network behaves synchronously. Throughout the subsection, 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 subsection, 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.

C.1 Good-Case Complexity

For the good-case complexity, we consider only executions where \( \text{GST} = 0 \) 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 \( n^2\kappa \) 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:

\[
\underbrace{O(\sqrt{n} \cdot \delta)}_{\text{DISPERAL}} + \underbrace{O(\delta)}_{\text{AGREEMENT}} + \underbrace{O(\delta)}_{\text{RETRIEVAL}} = O(\sqrt{n} \cdot \delta).
\]

Thus, the good-case latency of DARE is \( O(\sqrt{n} \cdot \delta) \).
C.2 DARE (and DARE-Stark) with Unknown $\delta$

To accommodate for unknown $\delta$, two modifications to DARE are required:

- **Disperser** must accommodate for unknown $\delta$. 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 $\delta$. Using the same strategy as for **Sync**, **Agreement** can tolerate unknown $\delta$. (The same modification makes DARE-Stark resilient to unknown $\delta$.)